2012 IEEE Sixth International Conference on Software Security and Reliability Companion

# Comparing Static Security Analysis Tools Using Open Source Software

2nd Lt Ryan K. McLean
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright Patterson AFB, OH 45433
Email: ryan.mclean@afit.edu

***Abstract*—Software vulnerabilities present a significant impediment to the safe operation of many computer applications, both proprietary and open source. Fortunately, many static analysis tools exist to identify potential security issues. We present the results of evaluating multiple subsets of open source code for common software vulnerabilities using several such static security analysis tools. These results aid other developers in better discerning which tools to use in evaluating their own programs for security vulnerabilities.**

**Keywords:** static analysis, software security, analysis tools, C/C++, open source

## I. Introduction

Modern programming languages afford programmers and software engineers great power when creating computer applications. Developers can wield memory management, multi-process control, and other operating system-level functionality. However, failing to properly use these mechanisms can quickly create security vulnerabilities in the hands of typical users and the malicious attackers. Static security analysis tools spot these problems before they become legitimate end-user issues, but can use relevant results for their projects. Because software capabilities can so easily be turned against software itself, developers must understand which tools are capable of identifying security flaws. This paper delivers several options for developers hoping to eliminate such vulnerabilities from their code by comparing such tools.

Most widely-used programming languages contain potential pitfalls that allow attackers to see, use, and even manipulate memory contents, execute arbitrary (albeit malicious) code, exploit race conditions, and more. Static security analysis tools provide software developers and testers with a variety of methods for quickly evaluating code for common vulnerabilities. Therefore, it is important that software developers understand how well these static analysis tools perform. This paper examines two static analysis tools, Flawfinder and RATS (Rough Auditing Tool for Security), and their ability to accurately detect security flaws in C code.

We present the results of running three code bases through two static security analyzers. Each analyzer produces different results, and each code selection has unique functionality. Developers can use this paper's approach as a pattern for critiqueing other such tools and code, or as the basis for selecting a static security analysis to use on their project.

Similar work exists in the area of static security analysis tool comparisons (e.g. [1]). Other related studies characterize or discuss the purpose of static analyzers [2], [3], how to effectively use them [4], or individual analysis tools and techniques [5]–[8]. This paper, however, focuses exclusively on comparing tools used to analyze network connection-based applications. Emphasizing this application subset provides deeper static security analysis tools knowledge to the community developing software of this genre.

This report details the approach we take for comparing static security analysis tools. Initially, we address static security analysis as a whole, including vulnerabilities and tools currently employed in practice. We also discuss why open source software likely contains security flaws, and we list the software tested in this experiment. The experimental setup explains factors considered in choosing the targeted vulnerabilities, software language and applications, and the actual analysis tools. Next, we elaborate on the process used to test the selected tools against code. The fourth section lists the experiment results, compares analyzer output and performance, and provides analysis tool suggestions.

## II. Background

### A. Static Security Analysis

*1) Common Vulnerabilities:* Buffer manipulation is prevalent in any moderately complex software. A buffer is an array used as a medium for transferring data to another array. Common buffer implementations include copying from one buffer to another, concatenating two buffers, reading user input directly into a buffer, and writing program output to the screen buffer. This process is conceptually simple, but is in fact the linchpin for one of the most common vulnerabilities - the *buffer overrun*.

The buffer overrun vulnerability allows an attacker to exceed the buffer's bounds. This can result in actions ranging from writing instructions to gaining full system access or control. Howard *et. al.* present the following code as an example of a buffer overrun vulnerability:

978-0-7695-4743-5 2012

DOI 10.1109/SERE-C.2012.16

```
#include <stdio.h>
void DontDoThis(char* input)
{
  char buf[16];
  strcpy(buf, input);
  printf("\%s\n", buf);
}
int main(int argc, char* argv[])
{
  // Length of argv[1] is unchecked
  DontDoThis(argv[1]);
  return 0;
}
```

If `argv[1]` is longer than `buf`, the stack grows by an unexpected amount when the program calls `strcpy()`. If this attack is correctly executed, the expected return address suddenly points to the attacker's new data. Therefore, if the attacker can recognize this vulnerability and subsequently manipulate the size and contents of `argv[1]`, there exists a very dangerous situation that developers can easily fix prior to release.

Another vulnerability commonly found in software is the *format string* bug. Howard, et. al. define the format string bug as "trusting user-supplied input without validation. [9]" In other words, the program uses user input as a password, echoes the program data or user input to the screen, or performs a similar action without checking the output data's validity. This action may seem harmless: `printf(data)` is perfectly allowable within the constructs of the C/C++ language.

For example, the `*printf()` family of functions expects a format string followed by matching arguments. Further, the `%x` format character in a C/C++ format string represents a hexadecimal value. However, if the calling code fails to supply a matching argument for any one format character, it uses stack contents regardless of the contents' intended purpose. This ensures that every format character receives a matching value, though not necessarily to the user's liking [9].

Herein lies the vulnerability. Suppose some code calls `printf(input)`, where the user supplies the unchecked value for `input`. If the attacker enters `%x`, the program effectively calls `printf(%x)`. Notice that no argument exists for output in hexadecimal format in place of the format character. To satisfy this condition, the `printf()` assembly code assumes that the format character's matching value is at the top of the stack. The code then outputs the stack's top value in hexadecimal form. This action grants the attacker access to stack operation, which can combine with other vulnerabilities (i.e. buffer overrun) to record and manipulate program behavior [9].

Developers also commonly use unsafe pseudorandom number generator (PRNG) functions under the presumption that the return value is *truly random*. In fact, many standard PRNG tools (C/C++'s `srand() and random()`, Java's entire `java.util.Random` package, and C#'s `Random` class) use deterministic algorithms to create *predictable* random numbers. If a program must implement a random number, best practice dictates using the system's cryptographic random number generator (CRNG). For example, Windows' CryptoAPI provides C and C++ programs access to the `CryptGenRandom()` routine [9].

Shell functions such as `system()`, `execv()`, and `popen()` start new program executions. While such functionality may be necessary, using it is unsafe. If an attacker can either force their way into a shell function, jump to where the victim application executes a shell call, or manipulate shell function call arguments, the system will likely forfeit access and control soon thereafter. When possible, developers should replace shell function calls with safer, library-defined functions.

*2) Current Tools:* The National Institute of Standards and Technology (NIST) maintains a list of static security analysis tools on their website [10]. NIST's list contains tools that detect a myriad of vulnerabilities in a variety of languages. Table I lists three commercial proprietary static security analysis tools.

TABLE I
PROPREITARY STATIC SECURITY ANALYSIS TOOLS

| Tool | Language(s) | Vendor | Targets |
|---|---|---|---|
| cadvise | C, C++ | HP | Memory leak, null-pointer dereference, tainted file path data, and more. |
| Coverity Static Analysis | C, C++, Java, C# | Coverity | Flaws and security vulnerabilities - reduces false positives and false negatives. |
| Goanna | C, C++ | Red Lizard Software | Memory corruptions, resource leaks, buffer overruns, null pointer dereferences, C++ hazards, and more. |

Two more tools, RATS and Flawfinder, are open source and feature the ability to detect the same vulnerabilities. We use these tools to run this experiment. Section four elaborates on these selections.

### B. Open Source Software

The open source software trend has grown tremendously in recent years. Whereas proprietary software contains work owned by an entity, open source software is available for open community development. Both open source and proprietary software packages are available for a slew of duplicative applications (i.e. OpenOffice versus Microsoft Office). However, the distributed nature of open source development implies that standards, even though many standards are open source themselves, are harder to enforce. We use open source software because it presumably contains more vulnerabilities than commercially-produced software.

## III. METHODOLOGY

### A. Experimental Setup

The following sections detail our experimental steps. First, a set of vulnerabilities influences the targeted language. Next,

we discuss the logic behind choosing open source applications and the operating system on which we run the test. We also give criteria for selecting static analyzers. Finally, we present the method for executing this experiment.

*1) Targeting Specific Vulnerabilities:* The tools listed in the previous section detect many types of vulnerabilities, and we focus on the categories which we estimate have the highest potential to cause system compromise. Buffer overrun errors are a routine avenue by which an attackers assume complete system access or control. Format string errors potentially give malicious users access to private or hardware-level system information. Poor random number generation implementations fail to adequately protect "protected" information. Finally, a careless developer who haphazardly implements shell functions offers the effective hacker a direct route to manipulating the system. The ability to detect these four vulnerabilities is the basis for this experiment's comparison.

*2) Choosing a Vulnerable Language:* Many languages are available to investigate as potential security analysis targets. To eliminate the variable of analyzing multiple languages and therefore make the experiment more uniform, we use one language throughout the test, C.

The C programming language is extremely vulnerable. Many C functions contain no security mechanism. Although new secure versions replaced older, flawed implementations (i.e. `strncpy()` replaces the buffer overrun-prone `strcpy()`), many software developers continue to use unsafe functionality. Such use constitutes a "red flag" in software security analysis.

Ultimately, we choose C over other languages because any given C program likely contains vulnerabilities due to its known security weaknesses and its ability to directly access computer hardware.

*3) Selecting Test Software Packages:* Vulnerabilities often present themselves when exploited by a remote user. A network connection is sufficient to provide remote access. Many widely-used applications are both open source and network-centric. We use several such applications to demonstrate actual vulnerabilities in actual programs, as opposed to contrived, toy examples. Table III-A3 lists these tools and their descriptions.

| Application | Description |
|---|---|
| PuTTY 0.61 [11] | SSH and telnet client |
| Nmap 5.61 [12] | Network security scanner |
| Wireshark 1.6.4 [13] | Network protocol analyzer |

TABLE II
SELECTED OPEN SOURCE TEST APPLICATIONS

*4) Operating System:* Linux provides a framework upon which software is easily installed. Further, Linux is open source - open source applications are widely available for download.

*5) Filtering Static Security Analysis Tools:* Static security analysis tools come in many different flavors. Novak *et. al.* define a taxonomy of static code analysis tools [3]. Within this taxonomy, we search for tools on the following rules:

| Criteria | Filter | Reason |
|---|---|---|
| Rules | Security | Security is a must for performing analysis most relevant to this experiment. Some tools might use style- or interoperability-related rules, but searching for primarily security-based tools is the most direct approach. |
| Technology | Syntax | All targeted vulnerabilities a matter of syntax, i.e. insecure function calls. |
| Language(s) | C/C++ | C is the language of choice; C++ is a superset of C. |
| Releases | N/A | Not concerned with regular releases. |
| Input | Source code | We desire to download source code and analyze it without compiling. |
| Configurability | N/A | Only desire pre-packaged capabilities. |
| Extensibility | N/A | Not concerned with ability to extend tool past this experiment. |
| Availability | Free, open source | Commercial tools typically require a costly license undesirable for this experiment. |
| User experience | Command line UI | Command line implementations likely require the least dependencies on additional UI libraries. |
| Output | List, text | We desire an ability to quickly filter and sort textual or itemized results using `grep`. HTML and XML output only complicate this task. |

TABLE III
CRITERIA FOR FILTERING STATIC ANALYSIS TOOLS

Based on the criteria in Table III-A5, the Flawfinder and RATS (Rough Auditing Tool for Security) static security analysis tools are sufficient for this experiment.

Flawfinder (General Public License, version 2+) is a database-centric analysis tool that searches a program's source code for known C/C++ insecure functions. The downloadable program includes the database. Flawfinder uses this database of problematic function calls to detect and report possible vulnerabilities. Per the Flawfinder website, Flawfinder "primarily does simple text pattern matching (ignoring comments and strings)" [14]. Additionally, Flawfinder provides type- (buffer, format, etc.) and risk-based (ranked zero, or lowest risk, through five, or highest risk) results categorization and a suggestion on how to fix the purported vulnerability. Finally, Flawfinder summarizes its findings according to total hits, hits at each risk level, cumulative hits at each risk level, and the cumulative nubmer of hits per thousands of lines of source code at each level [14].

RATS (GNU Public License) provides an approach similar to that taken by Flawfinder. Its authors advertise RATS' ability to scan C, C++, Perl, PHP, and Python source code and detect buffer overflows and race conditions, among other major

vulnerabilities. Like Flawfinder, RATS also uses a database to root out vulnerabilites. However, RATS categorizes hits according to risk level (Low, Medium, High), but not on their type (as in Flawfinder), forcing the tester to manually sort vulnerabilities. Manual labor aside, RATS accomplishes the goal of detecting known and highly-dangerous software vulnerabilities [15].

### B. Testing Process

After downloading and installing each analysis tool and test application, test execution is straightforward. First, we run each tool on each software package's source code. Output automatically redirects to a file. We filter the output for the desired statistic based on the following `grep` expressions:

- `"Hits ="` - total number of hits for each trial;
- `"Hits@level "` - total number of hits at each risk level;
- `"Hits@level+"` - number of hits at that risk level plus the sum of hits at all higher risk levels; and
- `"Hits/KSLOC@level+"` - Hits@level statistic per 1,000 lines of source code; and
- `"\[[1−5]\] ("` - all detected vulnerability message lines.

To flesh out RATS' data, we use one `grep` expression, `"High|Medium|Low"`, to grab all output messages. This is because every message detailing a vulnerability is on the same line as the risk level.

We used these command outputs to organize the data into spreadsheets to discern data trends. After sorting the data on function, we are able to quickly measure the number of detected vulnerabilities for each test software package.

One disadvantage of RATS is that it does not organize function calls by type. For example, Flawfinder notes `char[]`, `memcpy()`, and `strcpy()` as all being buffer overrun weaknesses. RATS does not classify past the function level, so we manually map RATS' discoveries to the desired categories. Fortunately, RATS flags nearly all of the same functions as Flawfinder, so creating the mapping, although time-consuming, is only a minor inconvenience.

## IV. Results and Discussion

In this section, we compare the two tools according to their ability to detect and report vulnerabilities. First, we discuss the post-analysis output each tool provides and the output's effectiveness at identifying and quantifying vulnerabilities. Next, we analyze how frequently each tool finds certain vulnerabilities. We also give brief consideration to each tool's reliability. Finally, we summarize the tools' overall performance and state our recommendation based on this experiment.

### A. Output Helpfulness

For this comparison, we choose to sample the output of analyzing PuTTY-0.61's `cmdgen.c` source file because it contains a moderate number of vulnerabilities, thus providing a decent results cross section.

```
cmdgen.c:976: High: strcpy
Check to be sure that argument 2 passed to this
function call will not copy
more data than can be handled, resulting in a bu
ffer overflow.
```

Fig. 1. RATS output of a single buffer overrun vulnerability.

In the PuTTY source file `cmdgen.c`, RATS detects a buffer overrun vulnerability, categorized as "High" severity, on line 976 in the form of `strcpy()`. While RATS assesses severity and explains why the vulnerability is dangerous, it neither classifies the vulnerability type (in this instance, buffer overrun) nor provides specific guidance on how to resolve the security issue.

```
cmdgen.c:976:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copyi
ng to destination.
  Consider using strncpy or strlcpy (warning, st
rncpy is easily misused).
```

Fig. 2. Flawfinder output of a single buffer overrun vulnerability.

Using the same PuTTY source file as Figure 2, Figure 1 shows Flawfinder's similar output, categorized as severity four out of five. Flawfinder also provides a recommendation for how to improve this statement's security. We show this output below in Figure 2.

```
Total lines analyzed: 1599
Total time 0.002061 seconds
775836 lines per second
```

Fig. 3. RATS analysis summary.

While RATS' brevity is useful, it could contain more helpful information (number of vulnerabilities per severity level, number of each vulnerability type, etc.). RATS accounts only for the total lines analyzed (no SLOC count) and how quickly it scanned the file.

```
Hits = 38
Lines analyzed = 1598 in 0.57 seconds (21362 lin
es/second)
Physical Source Lines of Code (SLOC) = 1081
Hits@level = [0]   0 [1]   8 [2]  13 [3]   0 [4]
  17 [5]   0
Hits@level+ = [0+]  38 [1+]  38 [2+]  30 [3+]  1
7 [4+]  17 [5+]   0
Hits/KSLOC@level+ = [0+] 35.1526 [1+] 35.1526 [2
+] 27.7521 [3+] 15.7262 [4+] 15.7262 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerab
ility.
There may be other security vulnerabilities; rev
iew your code!
```

Fig. 4. Flawfinder analysis summary.

Flawfinder produces verbose, detailed output for each file analyzed, as shown in Figure 4. In addition to reporting the number of vulnerabilities detected in the source code, Flawfinder also performs a Source Lines of Code (SLOC) count. Using its severity measurement system, it displays the number of hits per level, cumulative hits per level, cumulative hits per 1,000 SLOC per level, and the minimum risk level detected.

### B. Uncovered Issues

Both tools consistently discovered buffer overrun vulnerabilities. Even though secure methods exist which allow developers to safely manipulate buffers, unsafe function calls such as `memcpy()`, `strcat()`, and `sprintf()` exist throughout all three software packages. Format string bugs, primarily produced by the `*printf()` family, also exist in large quantities. All three software packages implement either `srand()` or `random()`, both of which are predictable and insecure "random" number generators. Finally, multiple vulnerable shell accesses occur throughout the test software.

*1) Buffer Overrun:* In each trial, Flawfinder detects more buffer overruns than RATS. Both analyzers detect buffer overrun violations by recognizing unsafe `string` library function calls, but Flawfinder goes an extra step by matching `char[] or WCHAR_T[]` declarations. While this practice surely highlights more *potential* vulnerabilities, a character array has a fixed length. The true vulnerability lies in the attacker's ability to access character arrays and write outside their bounds. Overall, both tools succeed at finding possible buffer overrun vulnerabilities.

Of the 2,930 buffer overrun vulnerabilities detected by Flawfinder in all three programs, nearly 17% (495) were `char[]` vulnerabilities. Unless these buffers are directly accessible to user input, these items pose little risk to overall security. A smaller fraction resulted from `memcpy()` calls with low risk similar to that posed by `char[]` buffers. Many dangerous `sprintf()` calls exist in PuTTY's code. With little effort, developers can transform these vulnerabilities into `snprintf()`, a size-controlled version of the former. The remaining buffer overrun detections (1,140 in total, or over one-third of all buffer overrun vulnerabilities) are from the `string` library.

RATS detects only 37% of the buffer overrun vulnerabilities found by Flawfinder, even though both tools detect the same types of insecure function calls. This is because RATS' intelligent filtering mechanism estimates true vulnerabilities much more aggressively than Flawfinder's similar mechanism. Specifically, RATS detects 1,099 total vulnerabilities. In total, 763 of those flaws are of the fixed-size buffer type detected as `char[]` vulnerabilities in Flawfinder. Further, RATS only describes 91 `string` library functions as vulnerabilities.

Fig. IV-B1. Potential Buffer Overrun Detections

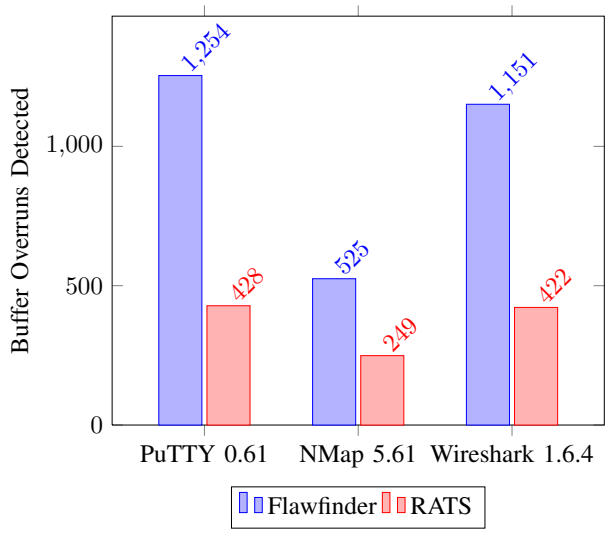
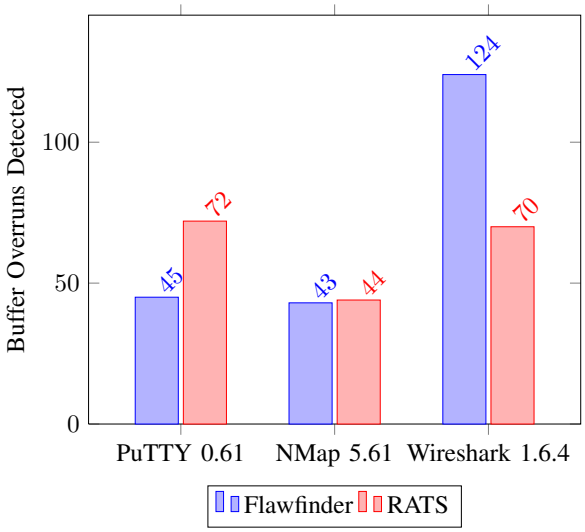



Fig. IV-B1. Format String Bug Detections




*2) Format String:* Whereas Flawfinder detects many more potential buffer overrun vulnerabilities than RATS, the two keep much closer tallies in finding format string bugs. We show this statistic in Fig. IV-B1. All format string bugs derive from potentially unsafe usage of `printf()`, `fprintf()`, `vprintf()`, or `vfprintf()`. Flawfinder highlights 212 vulnerabilities versus RATS' 186. However, each tool contains its own mechanism (as with buffer overruns) to make an educated guess as to whether a detection is, in fact, a vulnerability. This black-box logical process explains why the two produce virtually equal counts for Nmap, but vastly different numbers for the other two software packages.

*3) Random Numbers:* Although weak random number generation is sparse in these software packages, an attacker only needs one vulnerability. If the software uses these weak generators for cryptographic keys, such keys can easily be broken upon further system reconnaissance. The deterministic functions `srand()` and `random()` should be replaced by the host's secure cryptographic random number generator.

Fig. IV-B2. Weak Random Number Detections

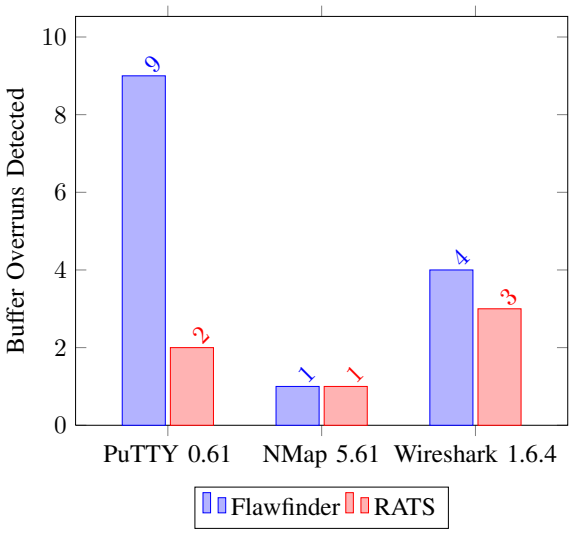
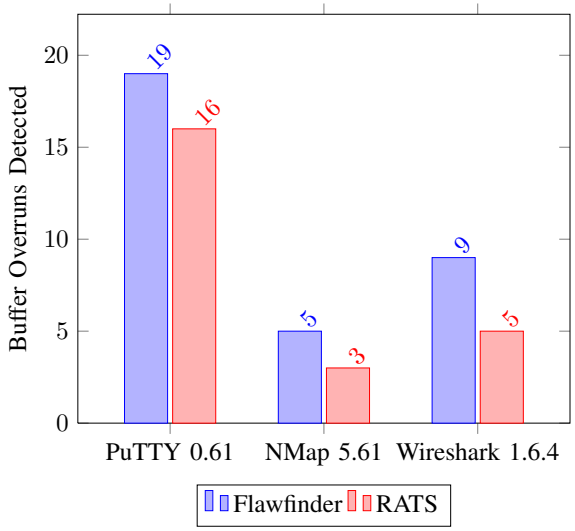



Fig. IV-B3. Shell Vulnerability Detections

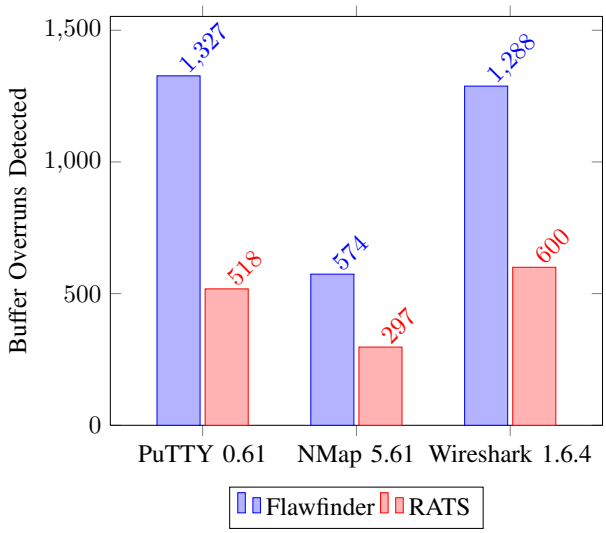



*4) Shell:* Shell function calls represent slightly more vulnerabilities than weak random number bugs. However, as shell calls represent direct access to the host system's intimate working details, these findings are significant. On the contrary, developers typically make these calls for a specific purpose that cannot be achieved via any other means. Therefore, the numbers in Fig. IV-B3 serve as a warning to limit the use of such functions.

### C. Vulnerabilites Detection Summary

The overall vulnerabilities tally in Fig. IV-C shows that Flawfinder produced 3,189 total vulnerabilities to RATS' 1,415 hits. While both tools detect false positives that pose no immediate security risk, RATS fails to cover all of the additional risky functions contained by Flawfinder's database. Furthermore, additional testing revealed that Flawfinder detects such vulnerabilities as integer vulnerabilities, obsolete functions, race conditions, and unsafe cryptographic operations.

Fig. IV-C. Total Vulnerabilities Detections




### D. Tool Usage Recommendation

In addition to simple function name matching, both tools use pattern-based logic to determine whether a vulnerability actually poses a security threat. Although Flawfinder reports 2.25 times the vulnerabilities discovered by RATS, it is not clear whether RATS more reliably discerns false positives, or if RATS simply misses many of the vulnerabilities Flawfinder detects. Flawfinder's ability to convey valuable information to the user is superior to that of RATS. Therefore, when choosing between Flawfinder and RATS, we recommend Flawfinder.

## V. Conclusion

Software security vulnerabilites pose a signficant threat to safe, trustworthy software operation. Modern static analysis tools provide developers with a means for extensively testing code for known flaws including buffer overruns, format string bugs, weak random number generation, system access via shell calls, and more. Tools such as RATS and Flawfinder detect these insecure design patterns. By using these tools and others, developers put themselves in a better position to defend against attacks and retain their users' trust.

## VI. Future Work

### A. Fix Vulnerabilities

The first step to expanding on this research would be to fix these vulnerabilities. Certain discernable patterns (i.e. function calls, algorithms, etc.) make static security analyzers possible. These patterns could conceivably form the basis for a tool or tool suite that automatically corrects these vulnerabilities.

### B. Expand Test Set

The software examined in this experiment represents only a fraction of all publicly available vulnerable software. Numerous types of vulnerable software exist, such as web applications, online gaming software, email clients, and more. Performing static security analyses of these and other software application genres would serve to inform otherwise-unaware developers about potential vulnerabilities in their area of work.

### C. Use Additional Analysis Tools

Testing with more tools would undeniably present a much more comprehensive assessment of both analysis tool capabilities and software security vulnerabilities. In addition to open source tools, and with the ability to purchase proprietary software, one could use industrial-grade analysis tools such as HP's cadvise, Coverity's Coverity Static Analysis, and Red Lizard Software's Goanna in a test similar to the one conducted in this experiment.

## ACKNOWLEDGEMENT

The views, opinions, and conclusions expressed in this paper are those of the author and should not be construed as an official position of the Department of Defense, the United States Air Force, or any other governmental agency or department.

## REFERENCES

[1] M. Mantere, I. Uusitalo, and J. Roning. "Comparison of Static Code Analysis Tools," in *Third International Conference on Emerging Security Information, Systems, and Technologies, 2009. SECURWARE '09.*, 2009, pp. 15-22.

[2] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, vol. 2, pp. 76-79, Nov.-Dec. 2004.

[3] J. Novak, A. Krajnc, and R. Zontar. "Taxonomy of static code ananlysis tools," in *MIPRO, 2010 Proceedings of the 33rd International Convention*, 2010, pp. 418-422.

[4] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh. Using Static Ananlysis to Find Bugs. *IEEE Software*, vol. 25, pp. 22-29, Sept.-Oct. 2008.

[5] D. Baca. "Identifying Security Relevant Warnings from Static Code Ananlysis Tools through Code Tainting," in *ARES '10 International Conference on Availability, Reliability, and Security 2010*, 2010, pp. 386-390.

[6] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, vol. 9, pp. 131-152, Spring 1996.

[7] D. Wagner, J. Foster, E. Brewer, and A. Aiken. "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of the Year 2000 Network and Distributed System Security Symposium (NDSS)*, 2000, pp. 3-17.

[8] J. Viega, J.T. Bloch, Y. Kohno, and G. McGraw. "ITS4: a static vulnerability scanner for C and C++ code," in *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference.*, 2000, pp. 257-267.

[9] M. Howard, D. LeBlanc, and J. Viega. *24 Deadly Sins of Software Security*. McGraw-Hill, 2010, pp. 90-93, 110-114.

[10] "Source Code Security Analyzers." Internet: http://samate.nist.gov/ index.php/Source_Code_Security_Analyzers.html, May 24, 2011 [November 15, 2011].

[11] "PuTTY: a free telnet/ssh client." Internet: http://www.chiark.greenend.org.uk/ sgtatham/putty/, November 27, 2011 [November 30, 2011].

[12] "Nmap - Free Security Scanner For Network Exploration & Security Audits." Internet: http://nmap.org/, n.d. [November 30, 2011].

[13] "Wireshark - Go deep." Internet: http://www.wireshark.org, n.d. [November 30, 2011].

[14] "Flawfinder Home Page." Internet: http://www.dwheeler.com/flawfinder/, n.d. [November 30, 2011].

[15] "HP Fortify — Rats - Rough Auditing Tool for Security." Internet: https://www.fortify.com/ssa-elements/threat-intelligence/rats.html, n.d. [November 30, 2011].