SREP+SAST: A Comparison of Tools for Reverse Engineering Machine Code to Detect Cybersecurity Vulnerabilities in Binary Executables

Thomas Ryan Devine
West Virginia University
Morgantown, WV
thomas.devine@mail.wvu.edu

Maximillian Campbell
West Virginia University
Morgantown, WV
mbc0034@mix.wvu.edu

Mallory Anderson
West Virginia University
Morgantown, WV
mka00009@mix.wvu.edu

Dale Dzielski
West Virginia University
Morgantown, WV
dale.dzielski@mail.wvu.edu

***Abstract*—Cybersecurity of mission components is vital for safety-conscious industries. This paper examines the effectiveness of using existing software reverse engineering products (SREPs) to first reverse engineer binary executables and then detect cybersecurity vulnerabilities through static application security testing (SAST) on the output (SREP+SAST). We analyzed 2.3 million lines of code from test suites and open source software. Results showed that SREP+SAST revealed 48% of the 20,129 vulnerabilities detected by SAST on the source code, including 35% of high-risk vulnerabilities (HRVs). We introduce Smaug, a novel, open source, customized SAST ruleset optimized for HRV detection. Smaug boosted our our best-performing combo by 20% and increased the HRV detection rate to 55%. For further validation, we traced vulnerabilities in source code linked to specific CVEs and found that Smaug improved the detection rate for CVE-specific vulnerabilities by 67% for our best-performing combo. Our methods and code are publicly available and we believe that further improving SREP+SAST could lead to enhanced security for systems that depend on COTS technologies.**

***Index Terms*—software reverse engineering, cybersecurity, vulnerability detection**

I. Introduction

The incorporation of Commercial Off-The-Shelf (COTS) software has long been known to pose security risks to safety-critical systems. However, the potential costs vs. benefits of utilizing third-party, COTS products are often significant and difficult to ignore. For example, as the satellite and space industries have evolved over the last 20 years, launching satellites into orbit is no longer the sole domain of wealthy nation-states, but includes both large and small businesses, universities, high schools, and even hobbyists [1]. This push toward space by non-traditional actors is largely driven by the affordability and pervasiveness of COTS technologies, which are likely “riddled with security vulnerabilities” [2].

Software vulnerability analysis can be accomplished by performing Static Application Security Testing (SAST), which involves static code analysis of the software’s source code [3]. If the source code is unavailable, however, SAST cannot be performed. Software Reverse Engineering (SRE) is the process of working backwards from binary, executable code to reconstruct the probable source code that was used to create it. Unfortunately, SRE is an inexact science, and although several SRE products (SREPs) exist that can effectively accomplish the task, they have recently been shown to produce dissimilar source code when given the same input [4].

In this paper, we empirically evaluate the effectiveness of SREP+SAST, a methodology for discovering cybersecurity vulnerabilities in machine code by first reverse engineering the machine code and then performing SAST on the reverse engineered machine code. We also compare the effectiveness of four existing SREPs and introduce a novel augmentation to a SAST tool. We present the methodology and results of a series of experiments designed to evaluate the ability of four SREPs to create reverse engineered machine code (SREP code) that could be analyzed for cybersecurity vulnerabilities using traditional static code analysis methods. We organized our research goals into three research questions:

1) Can vulnerabilities be revealed by SREP+SAST?
2) Are the tested SREPs equally capable or incapable of revealing vulnerabilities?
3) Which vulnerabilities are revealed by SREP+SAST?

The experiments have five stages, further detailed in Section IV. In Stage 1, we collected source code examples exemplifying dangerous software errors from open-source products with known vulnerabilities and manicured test suites. Stage 2 consisted of performing SAST on all source code to create vulnerability reports for each source code example. For Stage 3, we compiled the source code into machine code and then reverse engineered the machine code using four SREPs. In Stage 4, we performed SAST on the SREP code as in Stage 2 to generate vulnerability reports for each SREP code example. Finally, in Stage 5 we performed a comparative analysis of the vulnerability reports from the original source code and the SREP code to answer our research questions.

Our results show that SREP+SAST, while not perfect, is a promising methodology for detecting high-risk vulnerabilities (HRVs) in the SREP code. Our experiments with existing tools for SREP+SAST resulted in HRV detection rates at best of 35%. We developed an augmented ruleset optimized for detecting HRVs in SREP output called Smaug. Using

SREP+Smaug improved the detection rate for HRVs to 55%, a 20% increase over SREP+SAST. HRVs are the most likely to be exploited in real-world scenarios to cause harm to an organization. Additionally, when manually verifying the detection of vulnerabilities related to specific, real-world CVEs in vulnerable open source software, we found that our best combo of SREP+Smaug correctly identified 80% of these CVE-specific vulnerabilities, a 67% increase over SREP+SAST.

The main contributions of this paper are:

- Extensive empirical verification of the effectiveness of the SREP+SAST methodology,
- Creation of Smaug, a novel augmentation to an existing SAST tool, that improves performance when detecting HRVs in SREP code, and
- Comparison of four existing SREPs.

The remainder of this paper is organized as follows: Section II presents a review of current related work. In Section III, we provide background information on the tools and datasets used. Section IV details our methods and in Section V we present the results of our experiments. Finally, Section VI provides concluding remarks and directions for future work.

II. Related Work

We review five papers from three categories: SAST tools for software vulnerabilities [5], [6], malware analysis [7], [8], and testing [9].

In 2021, researchers analyzing Java open source projects investigated which source quality problems could be detected by warnings from SAST tools: SonarQube, Coverity Scan, Better Code Hub, Checkstyle, and FindBugs. The precision of each SAST tool was measured as the ratio between the true positive source code quality issues identified and the total number of issues the tool detected. The researchers concluded that different SAST warnings covered different issues and therefore found different source code quality problems [5].

Other studies explored the common defects that can be detected by SAST tools and the number of false alarms from these tools that should be ruled out. Researchers used three top tools when performing tests: CodeSonar, Code Prover, and Bug Finder. Well-known defects, such as static and dynamic memory defects, were successfully detected. However, difficult defects existed even for the top performance tools. CodeSonar was best at detecting concurrency defects, while Code Prover was best at stack-related defects [6].

The author of a 2020 study used automated Python scripts in Ghidra to detect buffer overflows in vulnerable sinks. The Ghidra API was used to disassemble executables from the Juliet Suite. They focused on vulnerability sinks from the *libc* functions. After the creation of the Python script, it was observed in the results that buffer overflow detection method was successful for 95% of the test cases [7].

A 2015 study investigated the use of machine learning and SAST to predict buffer overflow vulnerabilities. The authors used SREPs IDA-Pro and ROSE Disassembler and BinAnalysis and VulMiner for SAST. Using the Waikato Environment for Knowledge Analysis (WEKA) open source machine learning tool, they tested four machine learning algorithms: Naïve Bayes, Multi-Layer Perceptron Network, Simple Logistics, and Sequential Minimal Optimization. On average, the models were able to predict 60% of vulnerabilities correctly, except for Naïve Bayes, which had the highest accuracy prediction of 85.32% [8].

Other researchers evaluated five modern SAST tools (ARCHER, BOON, PolySpace C Verifier, Splint, and UNO), along with open source vulnerable software that was comprised of various versions of SendMail, BIND, and WU-FTPD. These researchers' evaluations indicate that while state-of-the-art SAST tools can find real buffer overflow with security implications, warning rates are unacceptably high [9].

III. Background

A. Software Reverse Engineering

Software reverse engineering is the process of working backwards from executable machine code to try to recreate what may have been the original source code in a higher level language. A program that performs reverse engineering on executables works by decompiling the machine code into assembly language instructions and then inferring what the high level source code may have been that produced the machine code. Since compilers perform many optimizations, an exact determination of the original source code is highly unlikely, but the process can approximate the source code.

To answer our research questions, we performed SAST on the output from four SREPs: Ghidra, RetDec, JEB Pro, and IDA Pro. Two are open source (Ghidra and RetDec) and two are available commercially (JEB Pro and IDA Pro). Ghidra is a software reverse engineering suite of tools developed by the National Security Agency's Research Directorate. It is primarily GUI-based. The Ghidra framework includes high-end software analysis tools to analyze compiled code from different instruction architectures. Ghidra is capable of disassembly, assembly, decompilation, graphing, and scripting.

RetDec is a command line interface (CLI) tool. Unlike Ghidra, which decompiles machine code method by method, RetDec can operate on one input binary file and outputs all the decompiled source code into one file. RetDec is a decompiler which produces retargetable machine-code based on LLVM and is the only decompiler in our research that does not have a GUI, making it easier to automate with scripts. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

JEB Pro and IDA Pro are both GUI based. JEB is a disassembler/decompiler for primarily Android Package Kit (APK) applications. However, JEB has the ability to disassemble/decompile other applications, such as Executable and Linkable Format (ELF) binaries. JEB decompiles Dalvik bytecode to Java source code and x86, ARM, MIPS, RISC-V machine code to C source code.

In the literature, IDA Pro proved to be a consistent top performer for analyzing applications. IDA's decompiler behavior is a mixture of Ghidra and RetDec, as IDA is a GUI based application which writes all decompiled code to a single file. Hex-Rays currently maintains and produces IDA Pro, which performs automatic code analysis using cross-references between code sections, knowledge of parameters of API calls, and other information.

B. Static Application Security Testing

SAST is automated testing that examines source code directly to detect vulnerabilities. SAST is an effective way of revealing certain cybersecurity vulnerabilities in source code, but is unable to analyze machine code. After reviewing several other products, we chose FlawFinder as our primary SAST tool. For a thorough discussion of SAST tools, see [10].

FlawFinder[1] is a free and open source SAST tool created by David A. Wheeler. FlawFinder scans C/C++ source code for potential vulnerabilities using lexical scanning to find tokens (such as function names) that suggest likely vulnerabilities, estimates their level of risk, and reports the results. FlawFinder can scan source code without compilation, unlike many other SAST tools. This was necessary for our research because SREP code is not meant to be recompiled. FlawFinder is officially Common Weakness Enumeration (CWE)-compatible and has earned the Core Infrastructure Initiative's (CII) Best Practice "passing" badge. FlawFinder categorizes each CWE into one-word phrases, such as Buffer, Integer, Race, etc., to give a general idea of the vulnerability that was found. See the FlawFinder documentation[2] for a full list of CWEs detected by FlawFinder in our experiments and their one-word descriptions.

FlawFinder also allows customization, which enabled us to create Smaug. Smaug is a novel ruleset augmentation for FlawFinder optimized to detect vulnerabilities in the SREP code. As detailed in Section V, many vulnerabilities associated with specific CVEs were not detected by SREP+SAST with FlawFinder. Many of the CVE-linked vulnerabilities we detected were buffer overflows in *libc* methods like *strcpy*, *memcpy*, *sprintf*, or *syslog*. Because these *libc* methods are commonly exploited by attackers, compilers, such as GCC, have been optimized to protect applications that use these them by ensuring that the size of the source buffer is smaller than or equal to the size of the destination buffer. Because of this protection, decompiled vulnerable *libc* methods were translated into '*_chk*' syntax. For example, '*strcpy*' becomes '*__strcpy_chk*'. The '*_chk*' syntax is missed by FlawFinder because '*__strcpy_chk*' is not a real *libc* method. Smaug, our new ruleset for FlawFinder, includes the '*_chk*' syntax.

C. Vulnerable Source Code

We analyzed 26 open source software products that contained known vulnerabilities and two software test suites. One test suite was designed specifically for reverse engineering cybersecurity testing with 356 test cases [11]. The other test suite was designed for the detection of cybersecurity vulnerabilities in C/C++ code with 99 test cases [12]. Overall, the data set used for our final analysis encompassed 2,341,590 lines of code in 5,041 files.

[1]https://dwheeler.com/flawfinder

[2]https://dwheeler.com/flawfinder/flawfinder.pdf

We obtained the vulnerable open source software by searching the *exploit-db*[3] and *MITRE*[4] vulnerability databases. On *exploit-db*, there are two checkboxes, "verified" and "has app", that are enabled to find software that is vulnerable and ensure the correct version of the application was downloaded. The software studied included vulnerable versions of OpenSSL, bash, grep sudo, python, and Rsync (the full list could not be included for space concerns).

We also analyzed several smaller applications, including aeon, dnstracer, and wifirxpower. These applications had only a few thousand lines of code, compared to other applications, with over 100,000 lines of code that we analyzed. The size of these applications made them easy to analyze, while adding more real world open source vulnerabilities to the research project.

IV. Experimental Methodology

Our experiments were conducted in a five-stage process, illustrated in Figure 1.

Stage 1: Source Code Collection We first collected the known exploitable software, which included the 26 open source software products and 2 software test suites that we described in detail in Section 3.3.

Stage 2: Source Code SAST We then used FlawFinder, our chosen SAST tool, to scan and generate the CSV and HTML vulnerability reports and saved them for later comparison. We also compiled the source code into executable machine code.

Stage 3: Decompilation with SREPs Next, we decompiled the machine code generated in Stage 2 using the four SREPs. We provide more details describing how we accomplished this stage, as this step was quite involved and required the creation of custom scripts.

When decompiling open source software products with RetDec, we used a single command in the terminal to decompile the entire executable into one file. When decompiling test-suites using RetDec, we wrote a shell script to traverse, decompile, and write results for every test case.

Decompilation with Ghidra was more involved than RetDec. Ghidra is primarily a GUI decompiler. Originally, we used a pre-written Java program, *GhidraDecompiler.java*[5], created by Guillaume Valdon, to automate the process of obtaining Ghidra's version of the decompiled code. However, the program required both the memory location and method name to decompile the method. We initially created our own program, *DragonBreath.py*[6], to assist with the task. Our first version of *DragonBreath.py* obtained both the memory location and name of the method being decompiled. The information from *DragonBreath.py* was then piped into *GhidraDecompiler.java* and the results would be written to a specified text file. However, *GhidraDecompiler.java* had a large margin of error

[3]exploit-db.com

[4]cve.mitre.org

[5]https://gist.github.com/guedou/a358df609c80d9fdc1ec4c348129005b

[6]https://github.com/Soup-tech/DragonBreath

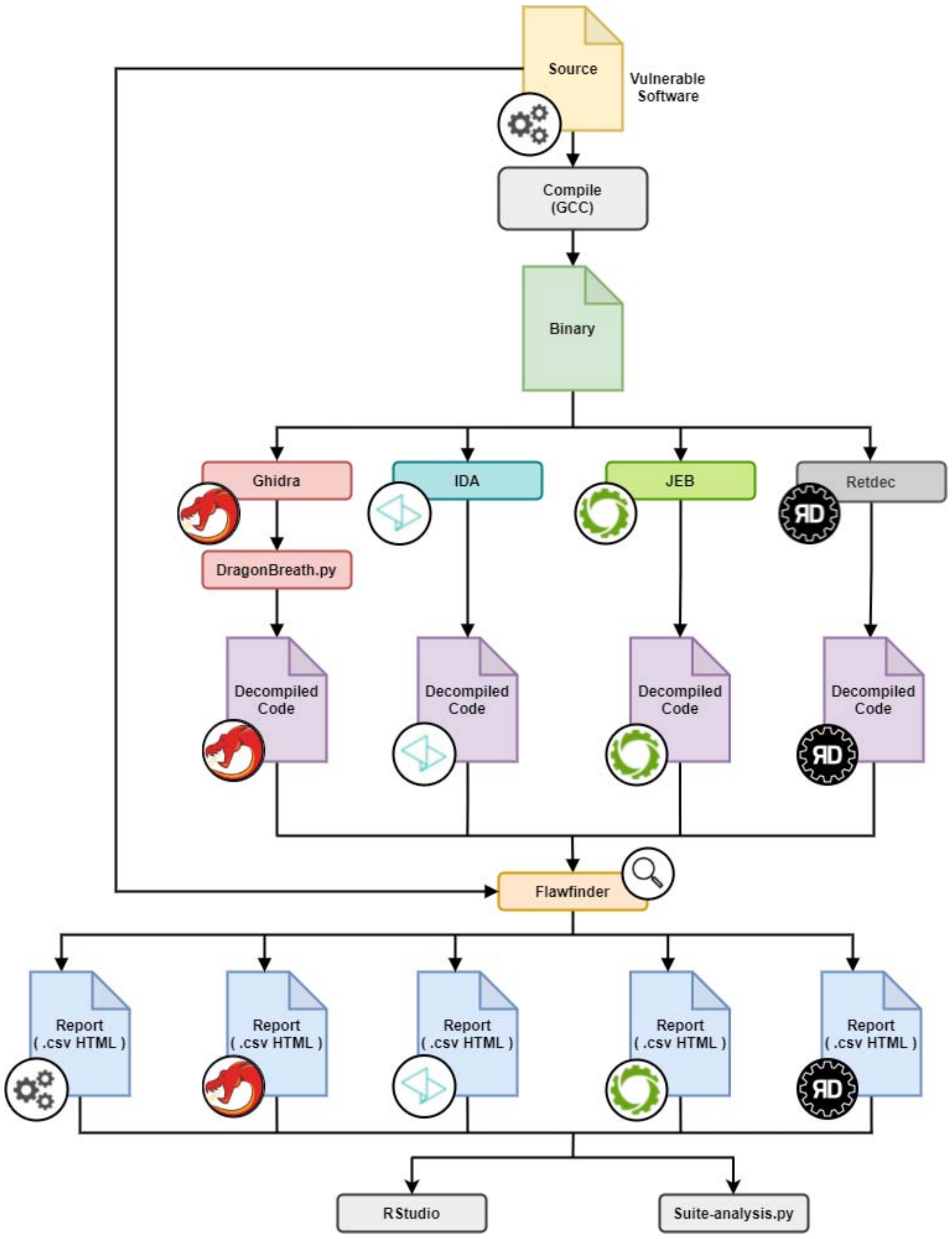

Fig. 1. Our scientific workflow depicting the experimental process used to evaluate all source code in this research.

and produced more empty text files than decompiled machine code. *DragonBreath.py* generated a list of memory addresses for Ghidra to decompile. However, the memory addresses were not always accurate and Ghidra would be unable to decompile the method that was referenced by the memory address. Also, if the method that was being decompiled was too large, *DragonBreath.py* would timeout and move onto the next method. Many of the bugs existed because Ghidra headless, the command line version of Ghidra, was used in *DragonBreath.py*. To overcome this, we wrote a revised version of *DragonBreath.py*, which served as a direct plugin for Ghidra. The inclusion of our plugin reduced the error created by *GhdiraDecompiler.java* to nearly zero. The time it took us to decompile an application using Ghidra reduced from hours to minutes. Since the older version of *DragonBreath.py* worked well with smaller applications, it was used when decompiling test suites.

Since JEB Pro and IDA Pro are similar to Ghidra, they also required some scripting to accomplish our goals. Fortunately, JEB Pro came packaged with a batch decompile feature.

JEB Pro's script worked similarly to *DragonBreath.py*, i.e., by decompiling every method in an application and writing the decompiled machine code to individual files. When we decompiled test-cases from the two test-suites, the CLI version of JEB Pro was used in conjunction with Python scripting. IDA Pro also came pre-packaged with a decompiler script. Unlike JEB Pro and Ghidra, IDA Pro writes all decompiled machine code into one file. Like JEB Pro, we wrote two small Python scripts to automate the process of decompiling and saving the decompiled machine code for every application in the test suits.

Stage 4: SREP code SAST We completed this stage by analyzing the SREP code produced by the four SREPs in Stage 3 with FlawFinder to generate CSV and HTML reports comparable to those produced in Stage 2.

Stage 5: Comparative Analysis Once we generated comparative reports for both the decompiled machine code and the original source code, we performed a comparative analysis of the results. To accomplish this, we placed all results into one CSV file for analysis and interpretation. This file contained all 42,805 vulnerabilities found throughout the course of this research project.

V. Results

This section provides the results of our comparative analysis of vulnerability reports generated during our experiments with SREP+SAST. We present these results as they relate to our original research questions, organized accordingly.

RQ1: Can vulnerabilities be revealed by SREP+SAST?

Results showed that it is possible to detect cybersecurity vulnerabilities in binary executables through SREP+SAST. However, the process with unaltered tools is only 47.6% effective overall, at best. Table I provides the total number of vulnerabilities detected throughout our experiments in the original source code (src) and in the SREP code. Using FlawFinder for SAST on the IDA Pro SREP code (IDA+FF), we were able to detect up to 47.6% of the total vulnerabilities present in the original source code. The performances of individual SREPs are compared under RQ2.

TABLE I
The total number of vulnerabilities and detection rates for the original source code (src) and the SREP code using FlawFinder (FF) and Smaug for SAST. The last two columns show HRV detections and detection rates.

SREP+SAST	*Total*	*DR (%)*	*HRV Total*	*HRV DR(%)*
src+FF	20,129	-	2,867	-
Ghidra+FF	7,382	36.7	846	29.5
RetDec+FF	2,291	11.4	393	13.7
JEB+FF	3,421	17.0	1,082	37.7
IDA+FF	9,582	47.6	999	34.8
Ghidra+Smaug	2,439	29.9	1,135	41.4
RetDec+Smaug	287	3.5	148	5.4
JEB+Smaug	1,360	16.7	869	31.7
IDA+Smaug	3,050	37.4	1,502	54.7

FlawFinder ranks vulnerabilities according to risk into categories from 1 to 5, with risk category 1 representing the least severe vulnerabilities and risk category 5 representing the most severe. In the last two columns of Table I, we show the total number of HRVs detected and associated detection rates. The results show that nearly 35% of HRVs can be detected from using SAST on SREP code. (Note that the high-risk detection rate results listed for JEB are anomalous, as described under RQ2.) Furthermore, when using Smaug, our optimized ruleset described in Section III-B, we were able to increase the detection rates for HRVs by almost 20%. Our best performing combo, IDA+Smaug, achieved a HRV detection rate of nearly 55%.

Figure 2, provides a breakdown of the number of vulnerabilities detected using FlawFinder in each risk category for all applications analyzed, separated out by SREP for each category and data set combination. The first row depicts the results for open source software, while the second and third rows show the results for both test suites. Note the floating scales for each row. The results show that a significant portion of vulnerabilities were risk category 1 and 2 for all data sets. risk category 1 and 2 represent code smells. Code smells are not vulnerabilities themselves, but rather violate principles of good software development, which negatively impacts the overall quality of the code, e.g., not NULL terminating any buffers that are initialized. Figure 2 also shows a large number of vulnerabilities detected in risk category 4. Vulnerabilities in this category have a risk of exploitation. Many of the CVEs that we examined were considered risk category 4 or higher, e.g., copying memory from a source to a destination buffer without any size restrictions or printing out a variable without a format identifier in a *printf* function call.

The majority of vulnerable *libc* methods detected by FlawFinder were risk level 1 and 2. FlawFinder is very thorough with buffer vulnerabilities. For example, every instance of a character array or buffer being instantiated or initialized was labeled as a potential vulnerability with risk level 1 or 2. Simultaneously, many of the critical vulnerabilities associated with CVEs were being missed due to the translation of the '*_chk*' syntax described in Section III-B. Smaug reduced the number of level 1 and level 2 vulnerabilities and increased detections of level 4 vulnerabilities. This is reflected in Figure 3 and results from our removal of buffer initializations and addition of all instances of vulnerable '*_chk*' syntax as discussed in Section III-B. While this revision to FlawFinder decreased the quantity of vulnerability detections, the overall quality of the vulnerabilities detected improved significantly, as we were able to achieve much higher detection rates on the HRVs that could lead to exploits.

RQ2: Are the tested SREPs equally capable or incapable of revealing vulnerabilities?

Ghidra and IDA Pro were consistently top performers when finding cybersecurity vulnerabilities regardless of SAST tool, as evidenced by the results presented in Table I and Figures 2 and 3. In particular, IDA+Smaug achieved detection rates for HRVs of almost 55%, making that combination the overall best performer, with Ghidra+Smaug in a close second place.

While JEB appears to do very well in the test suites,

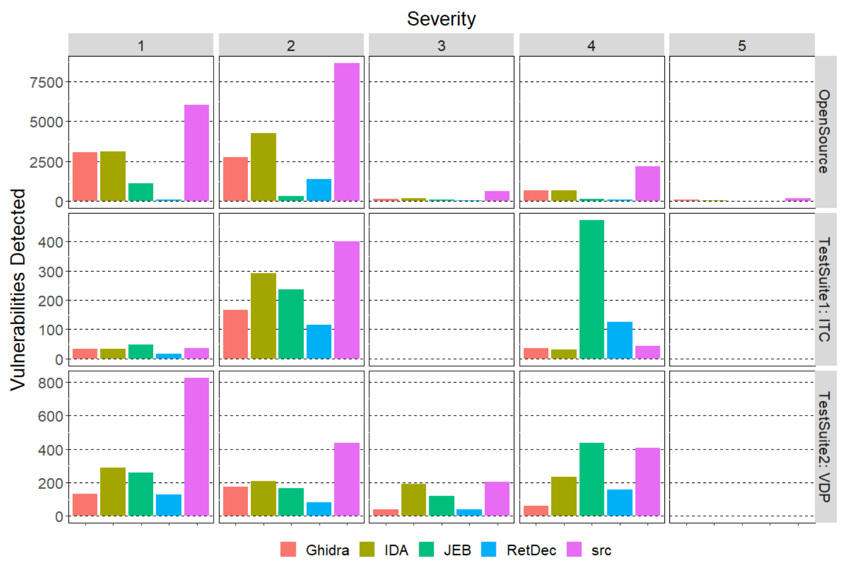

Fig. 2. Summary counts of vulnerabilities detected for the original source code (src) and all SREPs separated into columns by the risk of the vulnerability and rows by data set. Results shown are using FlawFinder for SAST.

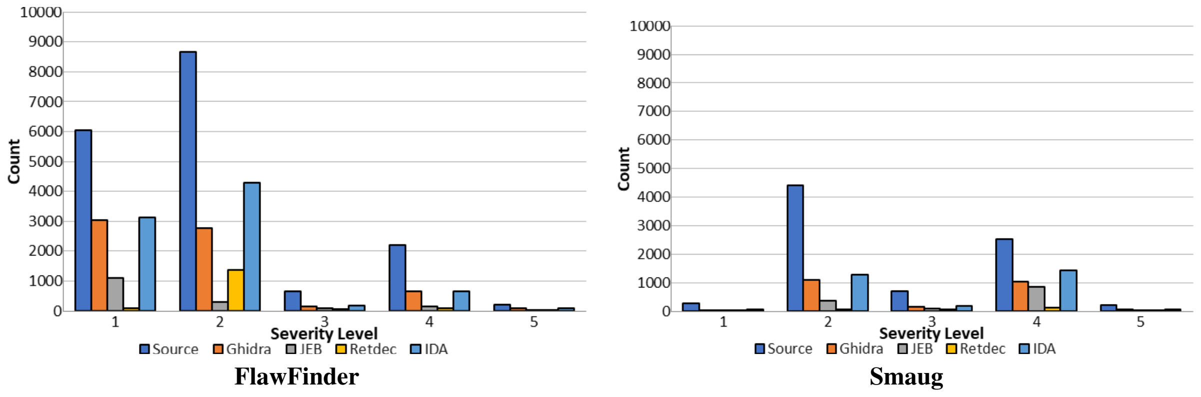

Fig. 3. The number and risk level of vulnerabilities detected by FlawFinder (left) and Smaug (right). Risk Level 1 represents the least severe vulnerabilities and Risk Level 5 represents the most severe vulnerabilities.

further inspection revealed that the results were anomalous. There should never be more vulnerabilities detected in the SREP code than in the source code, which can be seen for JEB in Figure 2. Upon examination, we found that JEB has an overabundance of Format vulnerabilities in the test-suites. This is due to test-suites using *printf* statements to categorize test-cases in the standard output. When decompiling machine code, JEB empties parameters to *libc* methods to save memory. Parameter-less *printf*'s are flagged as vulnerable by FlawFinder.

Additional problems for JEB include the incompleteness of its batch decompilation feature. This led to information loss and an excess of false positives, making JEB inconsistent and unreliable to use when batch decompiling. JEB is also primarily used for decompiling APKs, while all of the applications that we analyzed were ELF binaries. JEB, while consistent when decompiling individual methods in the GUI, lost information during batch decompilation which, in turn, created false positives from FlawFinder. This made JEB unreliable when paired with FlawFinder and also Smaug.

The poor performance of RetDec can be explained by its decompilation process. During decompilation, RetDec generalizes functions, *libc* methods, and character buffers. For example, rather than calling *strcpy* directly, RetDec creates a function which only contains *strcpy*. Every instance of *strcpy* encountered in the application will then call this function. When FlawFinder scans the decompiled code produced by RetDec, it will only see one instance of *strcpy* occur, no matter how many times it's wrapper function is called. This explains why RetDec was consistently the lowest performer out of the four SREPs.

Ghidra functioned the opposite of RetDec. Rather than generalizing much of the original source code, Ghidra attempted to be as precise as possible with its decompilation of the machine code. The only place where this precision falls short is with data types. For example, on a 32-bit architecture, integers are allocated 4-bytes in memory. When Ghidra decompiles an executable, it will only see that 4-bytes of memory have been allocated for a variable. Ghidra is unsure whether this is an integer or a 4 byte character buffer. Thus, Ghidra will label this variable as undefined. This was one reason why not as many results were found in Ghidra as there were with the original source code.

RQ3: Which vulnerabilities are revealed by SREP+SAST? The most detected vulnerabilities when performing SAST with FlawFinder on SREP code in this project were: *buffer*, *format*, *misc*, and *integer* vulnerabilities, as seen in Table II. Buffer overflow vulnerabilities constituted the largest portion of the vulnerabilities that were detected, by far. We attribute this in part to the fact that FlawFinder is vague with its definition of a buffer overflow vulnerability. However, since buffer overflow vulnerabilities are known to be one of the most common type of vulnerabilities to exist in software, it makes sense that FlawFinder would be thorough when scanning for buffer overflows.

TABLE II
VULNERABILITIES DETECTED IN ALL SREP CODE BY FLAWFINDER, WITH TOTAL COUNTS (CENTER COLUMN) AND CORRESPONDING PERCENTAGE OF THE TOTAL (RIGHT COLUMN).

Vulnerability	*SREP Count*	*Percent of Total (%)*
Access	102	0.4
Buffer	19,525	86.1
Crypto	85	0.4
Format	1,058	5.7
Free	6	0.0
Integer	679	3.0
Misc	754	3.3
Obsolete	13	0.1
Race	298	1.3
Random	42	0.2
Shell	93	0.4
Tmpfile	21	0.1

Table II also shows that all types of vulnerabilities were detected, to at least some degree. Figure 4, shows the detection rate achieved for each SREP on all vulnerabilities in the open source software as the percentage of the total number of vulnerabilities found in the SREP code vs the total number found in the original source code. We can see in Figure 4 that some vulnerabilities, such as Integer, Tmpfile, and Format, had very few or no discoveries in any of the SREP code, compared to the original source code, which had many.

We also note that a significant number of integer and format vulnerabilities were detected in the ITC Test Suite, particularly by JEB+FF. This is reflected in Figure 2 and is likely due to the large number of false positives FlawFinder produces when paired with JEB, as discussed above.

To verify which HRVs were detected by each combination of SREP+SAST, we manually traced methods associated with particular CVEs. In the original source code, we identified the usage of 46 vulnerable methods associated with 20 different

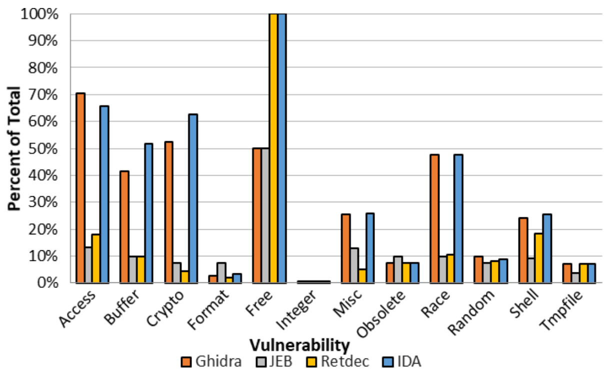

Fig. 4. The ratio of vulnerabilities found in the SREP code compared to vulnerabilities found in the original source code of the open source software described in Section III-C for each SREP.

CVEs. We inspected the vulnerability reports from each combination of SREP+SAST to search for detections we could clearly attribute to these 46 vulnerable methods. For 13 of the CVEs (and 16 associated methods), there was no difference in detection between FlawFinder or Smaug, i.e. either all of the SREP+SAST combos detected the vulnerability or all combos did not detect it. For the 30 remaining high-impact vulnerabilities, the results showed that using SREP+Smaug led to an average 48% increase in detections directly linked to CVEs for all SREPs. Figure 5 shows a comparison of the number of CVE-specific vulnerabilities detected for FlawFinder (left) and Smaug (right). Both Ghidra and IDA Pro benefited the most from Smaug, with Ghidra+Smaug and IDA+Smaug both detecting 24 of the 30 CVE-specific vulnerabilities, as compared to the four detections achieved by both SREPs when combined with FlawFinder.

VI. CONCLUSIONS

In this paper, we presented an empirical evaluation of SREP+SAST, a methodology for detecting cybersecurity vulnerabilities in binary executables by first reverse engineering the executable with a SREP and then performing SAST on the reverse engineered machine code. We tested this method by scanning both the original source code and the reverse engineered machine code from 26 open source software products with known vulnerabilities and two vulnerability detection test suites for twelve types of cybersecurity vulnerabilities. Our experiments evaluated the effectiveness of four SREPs: Ghidra, RetDec, JEB Pro, and IDA Pro, combined with the SAST tool FlawFinder and an augmented version of FlawFinder we named *Smaug*. Finally, we performed a comparative analysis of the results to determine the effectiveness of SREP+SAST, which tools were most effective, and which types of vulnerabilities were detected.

Our analysis shows that SREP+SAST can successfully discover cybersecurity vulnerabilities in binary executables. The results showed that Ghidra and IDA Pro were best able to maintain the detectability of vulnerabilities in source code when reverse engineering the executables. When paired with

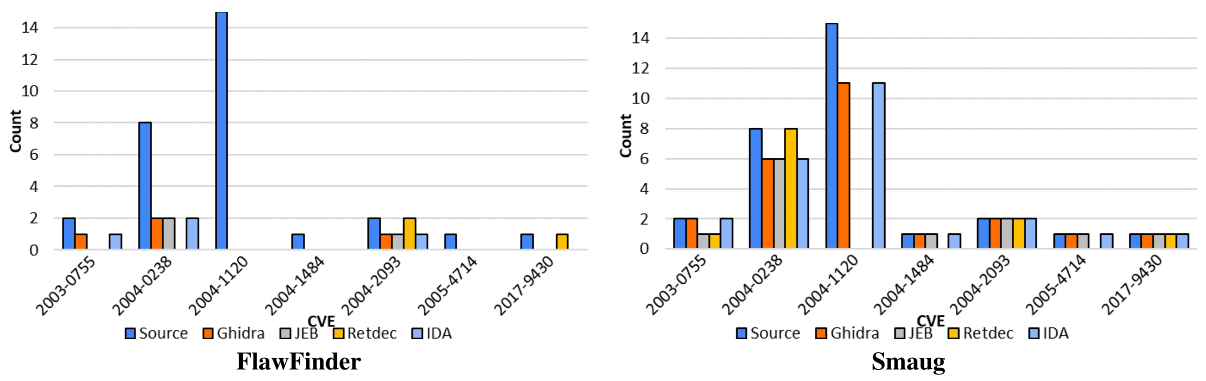

Fig. 5. The number of vulnerabilities detected per CVE by FlawFinder (left) and Smaug (right). Only CVEs with a different number of detections with each tool are shown.

FlawFinder, IDA Pro and Ghidra achieved detection rates for all vulnerabilities of 47.6% and 36.7%, respectively. When paired with *Smaug*, our optimized ruleset for FlawFinder, the HRV detection rates for IDA Pro and Ghidra (which are most likely to be exploited) increased to 54.7% and 41.4%, respectively. We manually validated SREP+SAST for the detection of cybersecurity vulnerabilities linked to specific CVEs in open source software and the results showed that Smaug increased detection rates of CVE-specific vulnerabilities for IDA Pro and Ghidra from 11.7% to 80%. Furthermore, we found that all twelve types of vulnerabilities examined were detectable in reverse engineered machine code to some degree, with buffer-related vulnerabilities being the most detectable, comprising 86.1% of all detected vulnerabilities.

In future work, SREP+SAST could be improved in a number of ways. Detection rates could be improved by customizing more detection rules for Smaug. Additionally, the results could be generalized further by testing different SAST tools and analyzing more vulnerable software. Finally, the results reported here use one compiler. It is important to explore the effect of different compilers, as each compiler performs different optimizations and may produce different results. In the future, we plan to include additional compilers in our analysis.

VII. Acknowledgements

This research was made possible by the NASA Katherine Johnson IV&V Facility through the WV Space Grant Consortium (Grant #80NSSC18M0128).

References

[1] B. Nussbaum and G. Berg, "Cybersecurity implications of commercial off the shelf (cots) equipment in space infrastructure," in *Space infrastructures: From risk to resilience governance*. IOS Press, 2020, pp. 91–99.

[2] G. Falco, "Job one for space force: Space asset cybersecurity," Belfer Center, Harvard University, Cambridge, MA, Tech. Rep., 2018.

[3] A. Brucker and U. Sodan, "Deploying static application security testing on a large scale," *Sicherheit 2014–Sicherheit, Schutz und Zuverlässigkeit*, 2014.

[4] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 475–487.

[5] V. Lenarduzzi, S. Lujan, N. Saarimaki, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," 2021, pre-Print.

[6] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," 11 2015.

[7] C. Wikman, Eric, "Static analysis tools for detecting stack-based buffer overflows," Naval PostGraduate School, Monterey, CA, Tech. Rep., 2020.

[8] B. M. Padmanabhuni and H. B. K. Tan, "Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, 2015, pp. 450–459.

[9] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: Association for Computing Machinery, 2004, p. 97–106.

[10] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Addison-Wesley Professional, 2007.

[11] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158

[12] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2015, pp. 12–15.