for applying *CCFinder*. First, since our tool is developed and operated under Linux, we apply only the back end of *CCFinder*. One challenge is that, since the default Linux version of *CCFinder* is designed to work on *Ubuntu 9*, the newer versions of many libraries are no longer valid for *CCFinder*. Therefore, several libraries need to be installed separately, e.g., *libboost-dev* and *libicu-dev*, which will depend on the specific version of the Linux system and can be determined based on the warnings and errors produced by *CCFinder*. Second, various parameters can be fine tuned in *CCFinder* to customize its execution mode [21]. In particular, the most important parameters include b, the minimum length of the detected code clones, and t, the minimum number of types of tokens involved. We have chosen $b = 20$ and $t = 8$ based on experiences obtained through extensive experiments. In addition, parameter w is used to determine whether *CCFinder* will perform inner-file clone detection whose results contain clones between different parts of the same software application, which is not our focus, and therefore w is set to be *f-w-g+* to focus on inter-file clones. Finally, the default output of the *CCFinder* is stored in a binary file with *.ccfd* extension. Since we do not install any front end of *CCFinder*, we apply the command *./$PATH/ccfx -p name.ccfd* to translate the *.ccfd* file into a human-readable version. The resultant file contains only the token information, which cannot be directly mapped back to the source code files. Therefore, we have developed a script, *post-prettyprint.pl* [37], to convert the token information into corresponding line numbers in the source code.

– *The Source Code Labeling Module* As mentioned above, the converted output of *CCFinder* provides only the file name and line number of the clone segments, without information needed for mapping them back to the original source code. For the purpose of generating traceable output with source code fragments, a mapping between the line number of the clone segments and the source code needs to be established. This second module is designed for this purpose by automatically retrieving a clone code segment from the source code according to the result of *CCFinder*.

– *The Visualization and CAS Calculation Module* The visualization module generates the results of clone segments. The results include clone ID, file path, function name, clone segment, start line number, and end line number. The visualized output is organized as an *XML* tree with labels. The label *contents* contains the source clone segments from *CCFinder* outputs. Label *funcname* reveals the function names corresponding to the clone segments, and label *io* contains the common I/O functions. To calculate the common attack surface, we first need to identify the I/O functions. In our experiments, we have obtained the list of I/O functions from the GNU C library [39] (glibc), which is the GNU project's implementation of C standard library, as the database for examining the entry/exit points. In total, 256 I/O functions are stored in our database, e.g., function *memcpy()* or *strcpy*, which could take user inputs as the source, and copy them directly to the memory block pointed to by the destination. Such functions have caused many serious security flaws including CVE-2014-0160 (i.e., the Heartbleed bug [7]). The final result of common attack surface is calculated based on the I/O functions shared among all software applications, and can be stored either in a file or into the database.

5 Experiments

This section presents experimental results on applying our tool *CASFinder* to real world open source software.

5.1 Dataset

To study the common attack surface among real world software applications, we need a large amount of open-source software to apply our tool. For this purpose, we have developed a script to automatically parse the download links at the open-source software hosts. Our research shows that *GitHub* [14] provides the customized API for users to search open-source software applications with customized requirements and to download them automatically. The results are presented in json code, which contains the download link of each application together with other information. In our experiments, we have set the parameter *language* to C programs, and use parameters *q*, *sort*, and *order* to specify the query conditions and to customize the sequence of results. We have developed the script to parse the json format output from the *GitHub* automatically and to store the information of the software download link, authors, publish time, size, and other descriptions into our local database. All the download links for each software application are stored separately. Since *Github* has a limitation with respect to the maximum requests in a certain amount of time, we design the process to sleep for certain time after each query. Our experimental environment is a virtual machine running Ubuntu 14.04, with the Intel core i3-4150 CPU and 8.0GB of RAM. We have applied our tool to totally 293 different software applications belonging to seven categories. The software applications belong to several categories as follows: 32 in Databases, 62 in Web servers, 25 in ssh servers, 79 in FTP servers, 41 in TFTP servers, 6 in IMAP servers, and 48 in firewalls. Those amount to totally $\binom{293}{2} = 42778$ pairs of software applications tested using our tool in the experiments.

5.2 Cross-Category Common Attack Surface

In this section, we apply the two proposed common attack surface metrics to totally 42,778 pairs of real world software. The first set of experiments reveal the existence of common attack surface between different categories of software applications. To convert the results to a comparable scale, we have normalized the absolute value of common attack surface reported by *CASFinder* by the size of the software. Figure 5 shows the existence of common attack surface across seven categories. The percentages on top of the bars inside each figure indicate the level of common attack surface between the category mentioned in the title of the figure and all the seven categories. We can observe that common attack surface exists in all of the category combinations. For example, the *DB* category has the highest level of common attack surface inside its own category (between different software inside that category), 27.9%, and it also shares more than 9% common attack surface with any other category.

In summary, the results across all categories are shown in the heat map in Table 1 where a darker color indicates a larger CAS value between the pair of categories. A visible diagonal with the darkest color in the heat map indicates the expected trend that

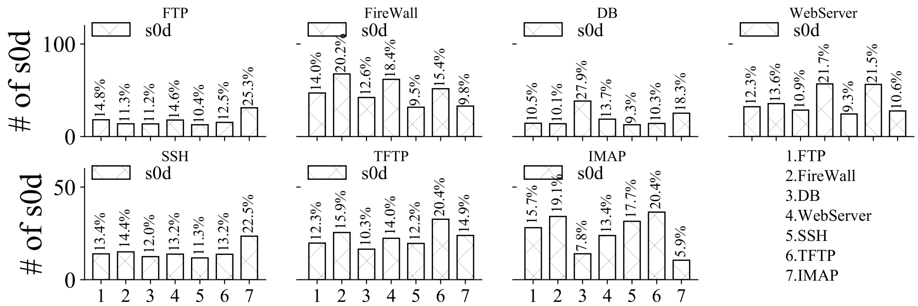


Fig. 5. Common Attack Surface across Categories

different software in the same category yield the highest level of common attack surface, most likely due to their similar functionality, except for *SSH*. In fact, the category *SSH* has the lowest level of common attack surface within its category. The reason is that the *SSH* category only contains 25 software applications, which is not sufficiently large to produce any reliable trend. Due to similar reasons, we have omitted the results from the *IMAP* category in the heat-map.

	FTP	FireWall	DB	WebServer	SSH	TFTP
FTP	18.2	13.8	13.7	17.9	12.8	15.3
FireWall	47.1	67.6	42.2	61.8	31.7	51.7
DB	14.4	13.9	38.4	18.8	12.8	14.1
WebServer	32.2	35.6	28.6	56.9	24.4	56.3
SSH	13.9	15.0	12.5	13.8	11.8	13.7
TFTP	19.8	25.5	16.5	22.4	19.6	32.6

Table 1. HeatMap for Common Attack Surface in Different Categories

After understanding the general existence of common attack surface among the seven categories of software applications, we aim to study more specific trends in our second sets of experiments. The left chart in Figure 6 shows the accumulated number of pairs of software applications in the absolute value of common attack surface. The figure depicts only the results with a nonzero value, which include totally 9,852 pairs (which amounts to about 1/8 of the total number of pairs). We can observe that the accumulated number of pairs of software applications increases quickly before the value of common attack surface reaches about 12 and afterwards the accumulation flattens out. About 20% of software share common clone segments, and 56% of the clone segments contain at least one common attack surface. The right chart in Figure 6 depicts the relationship between common attack surface and sizes of the software. We use the absolute values of common attack surface in this experiment. For the sizes, we use the normalized combined sizes $\log_{1000}(A^B)/1000$ when software A is compared with software B. We can observe that, with increasing sizes of the software, the value of common attack surface generally increases. This is as expected since the number of I/O functions would be roughly proportional to the size of the software.

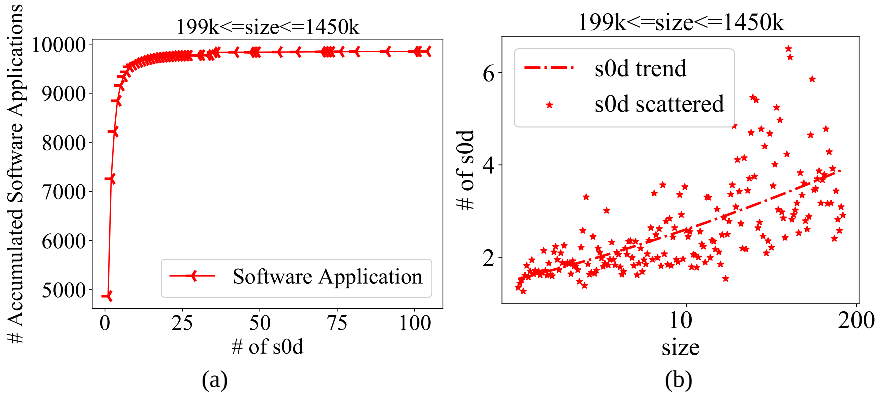


Fig. 6. CAS in Accumulated Software Application Pairs(a), CAS Trend vs Size(b)

The left chart in Figure 7 compares the average number of I/O functions and the average common attack surface over several years. The blue bars indicate the average number of I/O functions used in the software applications tested in our experiments based on the publishing year. The average number of I/O functions per software application does not have a simple trend and is used as a baseline for comparison. We can observe a clear downward trend in the average value of common attack surface over time, with software published around 2010 having a much higher value of common attack surface compared with more recent years, regardless of the number of average I/O functions. We believe this trend shows that code reusing plays a major role in common attack surface, since the trend can be easily explained by the backward nature of code reusing (i.e., programmers can only reuse older code). The right chart in Figure 7 explores the trend of the probabilistic common attack surface metric versus the size. The value of the probabilistic common attack surface metric decreases since the increase of the number of I/O functions in software applications is faster than the increase of common attack surface.

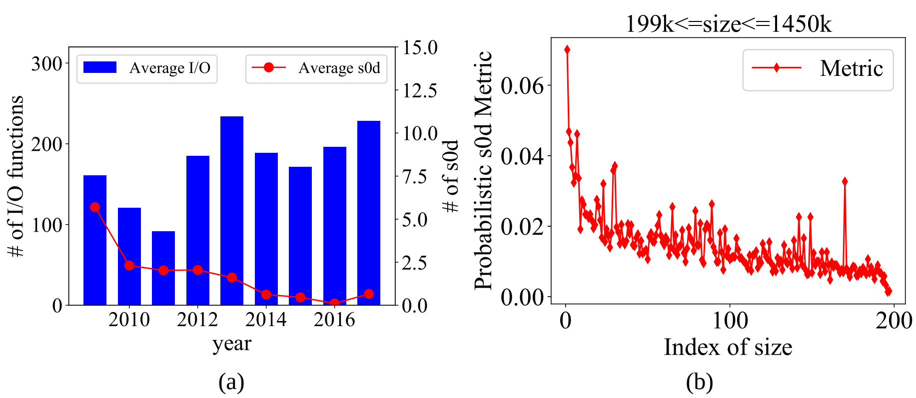


Fig. 7. CAS Trend in Years(a) and The Probabilistic CAS Metric(b)

In fact, those results match the results of existing vulnerability discovery models, which generally show that larger software applications typically have more vulnerabilities but a lower probability for having vulnerabilities per unit of software size. For example, Google Chrome (with the number of lines at 14,137,145 [1]) has 1,453 vulnerabilities over nine years [8], while Apache (with the number of lines at 1,800,402) has 815 over 19 years. However, the probability of having one vulnerability per unit of software size per year is $1.15 * 10^{-3}$% for Chrome and $2.4 * 10^{-3}$% for Apache (i.e., the larger Chrome has less vulnerabilities per unit of software size).

5.3 Common Attack Surface in the Same Category

We study the trend of common attack surface between software within the same category in this section. Figure 8 depicts the common attack surface for different sizes of software in the category *WebServer* and *FTP*, respectively, represented in both scattered and trending results. The orange scattered points and the dotted line indicate the result and the red dotted line is the same trend borrowed from Figure 6for comparison. We can observe that the trend of common attack surface in both categories increase with the size, which follows a similar trend as the cross category result. However, the trend of *WebServer* increases faster than the cross-category trend, which matches the results shown in Table 1. On the other hand, the trend in the *FTP* category grows slightly slower than the cross category trend, which can be explained by the fact that *FTP* shares a large amount of common attack surface with *WebServer* and *TFTP*.

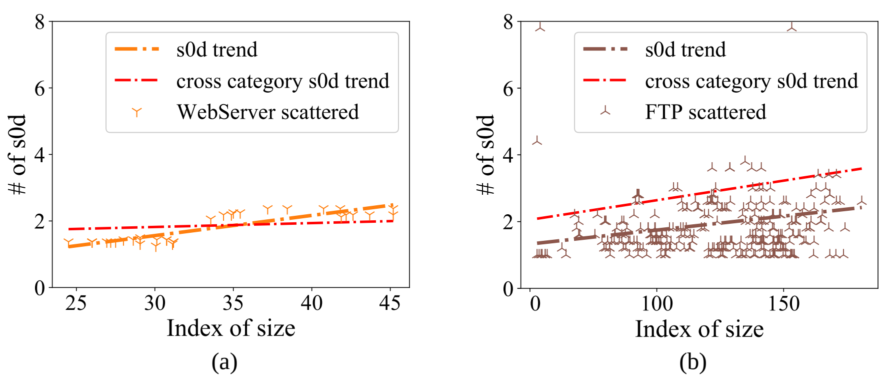
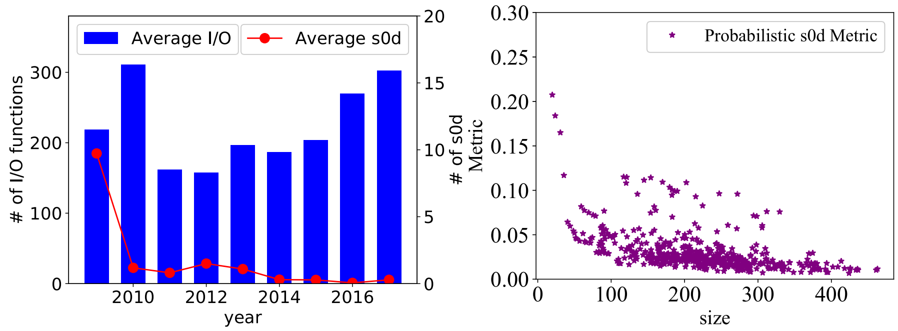


Fig. 8. Size Trend in Same Category, WebServer (a) and FTP (b)

The left chart in Figure 9 depicts the trend of common attack surface over time in the same category. Each blue bar represents the average number of I/O functions in the years in the same category of the experiments. The red line shows the average number of common attack surface in those years. Compared to Figure 7, the common attack surface in the same category has higher values, which also match the previous observations. The right chart in Figure 9 reveals the trend of the probabilistic common attack surface metric versus the size in the same category, which shows a similar trend as the cross category result, although the trend within the same category starts from a higher value around 0.20 (in contrast, the cross-category metric starts from 0.06).



(a) (b)

Fig. 9. Common Attack Surface Over Time and vs Size

6 Related Work

There exist extensive research on clone code detection although many of these tools are mainly for research purposes [41]. One of the popular tools in text-based clone detection is the *Dup* [2]; if two lines of code are identical after removing all whitespaces and comments, they are assigned as clone codes; the longest line matches are the output, but the minimum length of the reported code can be customized according to different needs. Another well-known approach [20] is applying the fingerprint in order to identify the redundancy on a substring of the source code. The fingerprinting calculation uses KARP-Rabins string matching approach [24, 25] to calculate the length of all *n* substrings. Ducasse developed [9] *duploc* which was designed to be a parsing free, language-independent tool which first reads the source file and sequences of the lines, then removes all comments and whitespace to create a set of condensed lines; afterward, a comparison is made based on the hash result, where scatter-plots indicate the visualization of a cloned result. Token-based clone detection is also one of the widely applied methods. One of the representative tools in token-based detection is *CCFinder* [22], which is applied in our work. Bakers Dup [2, 3] implements a similar approach as *CCFinder*. The detection process begins by tokenizing the source code, then using a suffix-tree algorithm to compare tokens. Unlike *CCFinder*, *Dup* does not apply transformation, but rather consistently renames the identifier. Raimar Falke [29] develops a tool called *iclones* [15], which uses suffix-trees to find clones in abstract syntax trees, which can operate in linear time and space. CP-Miner [31] as a well-designed token-based clone detector, uses frequent subsequence mining algorithms to detect tokenized segments. RTF [5] is a token-based clone detector that uses string algorithms for efficient detection; rather than using the more common suffix-tree, it utilizes more memory-efficient suffix array.

One of the leading tools using AST-based algorithm is the *CloneDR* developed by Baxter [6] which can detect exact and near-miss clone through applying hashing and dynamic algorithm. The *ccdiml* [38] developed by Bauhaus is similar to the *CloneDR* in the way of dealing with hash and code sequences, but instead of using AST, it applies IML algorithm in the comparing process. David and Nicholas [13] develop a tool named *Sim* which uses a standard lexical analyzer to generate a parsing-tree of two given software applications. The code similarity is determined by applying the maximum common subsequence and dynamic programming. One of the leading PDG-based tools is PDG-DUP presented by Komondoor and Horwit [26] and Komondoor and Horwitz's PDG-DUP [26] is another leading PDG-based detection tool, which identifies clones together and keeping the semantics of the source code to reflect software. As to metric-based clone detection, in [36] Mayrand uses the tool *Darix* to generate the metric and the clone identification is based on four values, which are name, layout, expression and control flow [36]. Kontogiannis [27] uses Markov models to compute the dissimilarity of the code by applying the abstract pattern matching. Five widely used metrics are applied in a direct comparison in [28]. There are also some other approaches that using hybrid clone detections. In [29], the authors apply the suffix trees to find clones in AST; this approach can find clones in linear time and space.

The concept of attack surface is originally proposed for specific software, e.g., Windows, and requires domain-specific expertise to formulate and implement [16]. Later on, the concept is generalized using formal models and becomes applicable to all software [34]. Furthermore, it is refined and applied to large scale software, and its calculation can be assisted by automatically generated call graphs [33, 32]. Attack surface has attracted significant attentions over the years. It is used as a metric to evaluate Android's message-passing system [23], in kernel tailing [30], and also serves as a foundation in Moving Target Defense, which basically aims to change the attack surface over time so to make attackers' job harder [18, 17]. The study on automating the calculation of attack surface is another interesting domain, e.g., COPES uses static analysis from bytecode to calculate attack surface and to secure permission-based software [4]. Stack traces from user crash reports is used to approximate attack surface automatically [43]. The correlation between attack surface and vulnerabilities has also been investigated, such as using attack surface entry points and reachability to assess the risk of vulnerability [46]. A study about the relationship between attack surface and the vulnerability density is given in [45], although the result is only based on two releases of Apache HTTP Server. Despite such interest in attack surface, to the best of our knowledge, the common attack surface between different software has attracted little attention.

7 Conclusion

In this paper, we have defined the concept of common attack surface and implemented an automated tool for evaluating the common attack surface between given software applications. We have conducted experiments on real open source software and examined the common attack surface both within and between software categories. Our results have shown common attack surface to be pervasive among software. Our work still has some limitations which will lead to our future work. First, since we rely on *CCFinder* our tool also inherits its limitations, and one future direction is to explore other clone detection tools. Second, we have focused on entry/exit points of attack surface, and one future direction is to also consider channels and untrusted data items. Third, we have focused on the C language in this work, and extending it to other languages with different entry and exit libraries is an interesting future direction. Finally, we plan to extend the effort on correlating between common attack surface and known vulnerabilities.

Disclaimer

Commercial products are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the identified products are necessarily the best available for the purpose.

References

1. Open hub. `https://www.openhub.net/`, 2017.
2. Brenda S Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.
3. Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.
4. Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM, 2012.
5. Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 513–516. ACM, 2007.
6. Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
7. Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. Heartbleed 101. *IEEE security & privacy*, 12(4):63–67, 2014.
8. CVE Community. Common vulnerabilities and exposures. `https://cve.mitre.org/`, 1999.
9. Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.
10. Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.
11. Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56, 2017.
12. AK Ghosh, D Pendarakis, and WH Sanders. Moving target defense co-chairs report-national cyber leap year summit 2009. *Tech. Rep., Federal Networking and Information Technology Research and Development (NITRD) Program*, 2009.
13. David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, volume 31, pages 266–270. ACM, 1999.
14. GitHub.Inc. A web-based hosting service for version control using git. `https://github.com`.
15. Nils Göde and Rainer Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 219–228. IEEE, 2009.

16. M. Howard, J. Pincus, and J. Wing. Measuring relative attack surfaces. In *Workshop on Advanced Developments in Software and Systems Security*, 2003.
17. S. Jajodia, A.K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. Springer, 2012.
18. S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, and X.S. Wang. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 1st edition, 2011.
19. Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.
20. J Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM*, volume 94, pages 120–126, 1994.
21. Toshihiro Kamiya. Tutorial of cli tool ccfx. `http://www.ccfinder.net/doc/10.2/en/tutorial-ccfx.html`, 2008.
22. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
23. David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 69–80. ACM, 2012.
24. Richard M Karp. Combinatorics, complexity, and randomness. *Commun. ACM*, 29(2):97–109, 1986.
25. Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
26. Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.
27. K Kontogiannis, M Galler, and R DeMori. Detecting code similarity using patterns. In *Working Notes of 3rd Workshop on AI and Software Engineering*, volume 6, 1995.
28. Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.
29. Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.
30. Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *NDSS*, 2013.
31. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.
32. P. Manadhata and J. Wing. An attack surface metric. Technical Report CMU-CS-05-155, 2005.
33. P. Manadhata and J. Wing. An attack surface metric. *IEEE Trans. Softw. Eng.*, 37(3):371–386, May 2011.
34. Pratyusa Manadhata and Jeannette Wing. Measuring a system's attack surface. Technical Report CMU-CS-04-102, 2004.
35. Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.
36. Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.
37. Petersenna. Ccfinder core. `https://github.com/petersenna/ccfinderx-core`.

38. Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus-a tool suite for program analysis and reverse engineering. In *Ada-Europe*, volume 4006, pages 71–82. Springer, 2006.
39. Trevis Rothwell. The gnu c reference manual. `https://www.gnu.org/software/gnu-c-manual/`, 2006.
40. Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.
41. Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541(115):64–68, 2007.
42. Apostolos Syropoulos. Mathematics of multisets. In *Workshop on Membrane Computing*, pages 347–358. Springer, 2000.
43. Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 199–208. IEEE Press, 2015.
44. David A. Wheeler. A program that examines c/c++ source code and reports possible security weaknesses. `https://www.dwheeler.com/flawfinder/`.
45. Awad A Younis and Yashwant K Malaiya. Relationship between attack surface and vulnerability density: A case study on apache http server. In *Proceedings on the International Conference on Internet Computing (ICOMP)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.
46. Awad A Younis, Yashwant K Malaiya, and Indrajit Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 1–8. IEEE, 2014.
47. Mengyuan Zhang, Lingyu Wang, Sushil Jajodia, Anoop Singhal, and Massimiliano Albanese. Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. *IEEE Transactions on Information Forensics and Security*, 11(5):1071–1086, 2016.

Appendix

Common Attack Surface and Vulnerabilities

We study the correlation between the common attack surface of two software applications and their shared vulnerabilities. Although, as mentioned earlier in Section 2.2, the concept of attack surface is not intended as a one-to-one mapping to actual vulnerability, the size of attack surface can still provide a rough indicator for the relative abundance of vulnerabilities, since entry and exit points represent the interfaces exposed by the software for accepting inputs from (or sending outputs to) the outside environment. Consequently, the common attack surface may also indicate shared vulnerabilities. Therefore, we study this correlation through experiments.

To evaluate the correlation between common attack surface and vulnerabilities, we examine pairs of software applications with respect to the results of a vulnerability scanner called *flawfinder* [44]. *Flawfinder* is an open-source tool that can be used to scan C and C++ source code and report potential vulnerabilities [44]. It is regarded as an effective tool for detecting misused functions with ranked risks. For the purpose of verifying the relationship between common attack surface and vulnerabilities, we manually compare our results with the results of *flawfinder*.

Our findings indicate that common vulnerabilities may indeed to some extent be correlated with common attack surface. For example, we examined the two software *SSH* and *simple-webserverche*, which are of the category *SSH* and *Webserver* applications, respectively. In *SSH*, the main file uses function *strcat* to copy data to an internal parameter *user_name*, and those data are applied without any boundary check. The same thing happens in the application *simple-webserverche* where a file named *server.c* calls function *handle_response()* to apply function *strcat*. The source parameter *curr_time* is applied before the boundary checking. Our tool successfully detects these code fragments as a common attack surface, while *flawfinder* reports that both have the potential to lead to similar *buffer overflow* vulnerabilities.

To further evaluate the extent of such correlation, we compare the outputs of *flawfinder* and the results of our tool, and Table 2 shows the level of correlation between the two. As the results show, in every category of software applications, there exist a certain percentage of vulnerabilities which correlate to the common attack surface.

FTP	4.07%	FireWall	3.74%	DB	3.40%	WebServer	4.08%	SSH	3.37%	TFTP	3.10%	IMAP	6.52%

Table 2. Percentage of Detected Vulnerabilities Which Correlate to Reported Common Attack Surface