Combinatorial Testing of Full Text Search in Web Applications

M S Raunak[1] D. Richard Kuhn[2] Raghu Kacker[2]
raunak@loyola.edu kuhn@nist.gov raghu.kacker@nist.gov
[1]Loyola University of Maryland [2]National Institute of Standards and Technology

Abstract: **Database driven web applications are some of the most widely developed systems today. In this paper, we demonstrate use of combinatorial testing for testing database supported web applications, especially where full-text search is provided or many combinations of search options are utilized. We develop test-case selection techniques, where test strings are synthesized using characters or string fragments that may lead to system failure. We have applied our approach to the National Vulnerability Database (NVD) application and have discovered a number of "corner-cases" that had not been identified previously. We also present simple heuristics for isolating the fault causing factors that can lead to such system failures. The test method and input model described in this paper have immediate application to other systems that provide complex full text search.**

I. INTRODUCTION

Web-based applications, especially ones driven by a back-end database, continue to be some of the most common custom software being developed in today's world. Many applications store their data in an SQL database or in a no-SQL data store at the back-end servers and query and update them to provide information searched by the users and other transaction-oriented service. The architecture of these applications is usually layered, utilizing many different, often loosely connected components. Testing these applications effectively remains a challenge for the software engineering community. High profile web application failures have resulted in cases where testing was insufficient [2][3]. A wide variety of general-purpose strategies and techniques for systematically testing applications are well established and widely practiced [1].

In this paper we focus on a highly specific test procedure for full-text search in the National Vulnerability Database (NVD), a large, heavily used public internet database. Text search is one of the fundamental components of web applications used in every industry. Thus, strong testing of this component is essential for high assurance, but it is often handled as simply one aspect of overall system testing. The test procedure documented in this paper seeks to provide stronger testing for this fundamental component.

After implementing a new feature in the NVD, developers discovered that certain special characters resulted in "Server Error" responses. However it was not clear which specific combinations of special characters triggered this response, or how many such problematic cases existed. Some of these were corrected, but it was decided to apply combinatorial methods to attempt to identify all inputs that caused the server error response.

Decision makers in industry and government rely on testing to determine readiness of systems for deployment, so defensible measures of test completeness are essential. Testing any reasonable application exhaustively is nearly always impractical. The two crucial questions testing researchers aim to study are how to select the test cases, and how many to select; i.e., when to stop testing. These questions become especially challenging for integration and system testing of any large application. The goal is to select test cases at the integration and system testing level such that the fault finding probability is maximized. One can try to achieve this goal by systematically covering a large section of the input space including many corner cases or unexpected values that may potentially cause failures.

Combinatorial Testing [4] has been shown to be an effective approach. In this paper, we demonstrate use of combinatorial approaches to develop test cases that systematically test important components of database-backed web applications. Our case study reveals that such systematic exploration of input space using covering arrays can be a very useful way of identifying failure scenarios that are otherwise not discovered.

Web applications are typically developed in a multi-tiered fashion. Fig. 1 shows a high level generic architecture of most of these applications. The outermost layer is the presentation layer, which provides the interface for client interaction. Application developers use HTML, CSS, and client side scripting languages to make the user interface intuitive and useful. The Middle layer is often served by an Application Server framework such as JBoss, WebSphere or Netweaver. These application servers are often built around common web servers such as Apache and IIS. Programmers develop their business logic and run their programs on these Application Server environments. At the innermost layer sits a DBMS such as mySQL, MS SQL, Oracle, DB2 etc. The application servers provide support for interacting with the DBMS through standard protocols like Java Database Connectivity (JDBC), Common Object Request Broker Architecture (CORBA) etc.

While developing a web application, there are programming modules that are developed at each level. Programmers often write stored procedures, which are

groups of SQL statements stored and executed in the RDBMS, i.e., at the innermost layer, to reduce server-client

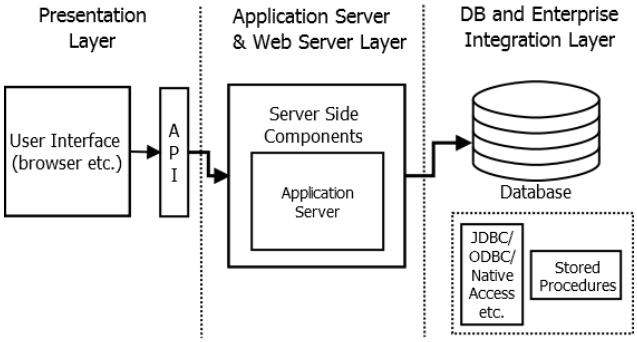

Fig. 1. A Typical Multi-Layered Web Application

network traffic, and to make the system more secure by hiding many of database-level details. The middle layer, where most of the business logic is implemented, often includes integration of other already developed modules or service calls to independent web services. Similarly, for receiving and reacting to user interaction and presenting the output of the application to the client, program modules are developed and deployed to interact with the middle layer directly. All these component interactions, both at the same layer and across layers, require systematic and thorough testing. Programmers and managers are usually good at making sure that unit tests are developed and run regularly while different programming units are being developed. However, testing unit interactions during integration, and testing the overall system systematically once it has been developed, often receive relatively little attention. Anecdotal evidence suggests that the primary reason for this relative lack of integration and system testing is often deadline pressure and not allowing enough time for testing during project estimation and planning phase. A second factor that makes this scenario more challenging is the fact that a systematic process of testing these types of application is not yet well established. A third challenge comes from the often dynamic nature of the underlying data, which makes it more difficult to develop test oracles to support test cases that can be automated.

In this study, we explore a systematic way of testing certain system level testing of database backed web applications and report the effectiveness of our approach.

II. Related Work

Because of their practical importance, database applications have been the subject of a variety of investigations of combinatorial test methods. In particular, combinatorial testing is especially appropriate because database systems must parse and interpret complex queries structured as regular expressions or predicates [15]. In addition to the complexity of the inputs, database applications are also characterized by the need to test both the database functions and the database system interaction with the application that accesses the database [20]. The test problem is further complicated by dependence on the initial database state, which may influence structural coverage metrics, i.e., the degree to which the code can be exercised [8] The complexity of database queries has led to the need for specialized coverage metrics that include the evaluation of conditions in search predicates [7].

Some research has shown that the distribution of *t*-way faults in MySQL database applications is similar to many other application domains, following the interaction rule that most faults are caused by a single factor or two factors interacting, with progressively fewer by 3-way or higher strength interactions [19]. The empirical data showed that a significant proportion of SQL faults involved 3-way or higher strength interactions, suggesting the need for combinatorial methods. Pairwise testing was shown to be effective [15] for discovering many bugs not detected by conventional test methods. It was shown that CT detected a wide range of previously undiscovered faults in a web based database using 2-way through 4-way testing [16]. These methods were also shown to be effective for testing security of database applications [17][18].

Also relevant to our work are investigations comparing the effectiveness of covering arrays with random test generation. This question has been studied in a variety of contexts. In many cases the comparison between combinatorial and random methods has considered only pairwise test arrays, with an equal number of randomly generated tests. Schroeder et al.[22] compared randomly generated tests with *t*-way arrays for $t = 2, 3, 4$. Covering arrays were generated using a tool called TVG, and applications tested had input model configurations of $2^{16}5^{1}8^{1}$ and $2^{7}3^{10}4^{2}$. Because the covering arrays were large, random test sets of the same size covered 95% to 99.99% of the *t*-way combinations, and there was no significant difference between *t*-way testing and random testing. This example illustrates the point that coverage of combinations is a key consideration, whether this is achieved by covering arrays or other test generation methods. Another study [23] found than manually constructed tests could be more effective than 2-way test arrays, but at higher strengths there was no difference, and results from randomly generated tests were not consistent. A number of studies have consistently found covering arrays to be more effective than random tests, including [24][25][26], which investigated the testing of logical expressions.

Others showing significantly better results for *t*-way testing include [27][28][29]. Two key considerations must be evaluated in comparing the two approaches to testing: combinatorial coverage of test sets, and input model design. It is easy to show that a large enough randomly generated

test set will cover a high proportion of *t*-way combinations, so the comparison between covering arrays and random test generation is largely a question of efficiency, at least at lower interaction strengths. For *t*-way testing of 4-way and above, a random test set covering the same proportion of combinations may be prohibitively large. The importance of the input model can be seen in research that demonstrates significant differences in structural coverage and fault-detection effectiveness as the input model is changed. Examples include [30], where branch coverage was increased from roughly 70% to 100% only through input model changes, and [31], which demonstrated improved fault detection results for both covering arrays and random tests depending on the input model used.

III. Approach

Our approach is to search for failure scenarios through systematic coverage of the input space and user interactions at the system testing level. For database backed web applications, especially the ones that primarily render some subset of data for information purposes, one of the major components is some sort of query functionality. Users are provided an interface to query the underlying information in many different ways. Consider the case of searching the catalogs of any library or the flight search in a travel application on the web. An important testing aspect in these scenarios is to verify that the search functionality behaves as expected under all circumstances.

In addition to testing for expected functional behavior, any application, especially the ones available over the web, also needs to be tested for potential failure scenarios against unexpected input. If there are unintentionally mistyped or maliciously created inputs that can cause system failure or unexpected behavior, then it also becomes a security issue, which demands attention. Developers and testers often test only the most common expected interaction from users, which is also commonly known as `Happy Path Testing'. This testing practice leaves out the necessary aspect of searching for system failures under unexpected user inputs or interactions.

We use a combinatorial approach and come up with covering array of a wide range of input combinations. We utilize these covering arrays to create test cases. For this particular study, we focus on primarily two types of test-scenarios: a) user inputted strings that may cause failures, and b) user selected options in a web form.

IV. Case Study

To apply our approach of developing effective test cases using combinatorial coverage for systematically testing database-backed web application, we selected the National Vulnerability Database or NVD [12] project. NVD is a project under the Computer Security Division of the National Institute of Standards and Technology (NIST). It maintains a repository of publicly known hardware and software vulnerabilities in a standardized fashion. Every vulnerability is uniquely identified by a CVE-ID (Common Vulnerability and Exposure Id), which is primarily assigned by the MITRE corporation and, to a limited degree, by some other CVE Numbering Authorities (CNAs) [13]. Once a reported vulnerability has been assigned a CVE-ID, it finds its way to the NVD group at NIST. Here the submitted CVE is thoroughly analyzed for standardization and is placed under one or more CWE (Common Weakness Enumeration) categories. Additionally, NVD analysts checks all the references, standardizes different aspects of the vulnerability description, and assign a severity score to the vulnerability following Common Vulnerability Scoring System or CVSS [14]. Once a new vulnerability has been standardized, categorized, and reference-checked, they are made available for public use through the NVD web site. This NVD data provides support for many valuable services such as enabling automation of vulnerability management, security measurement, and compliance (e.g. FISMA).

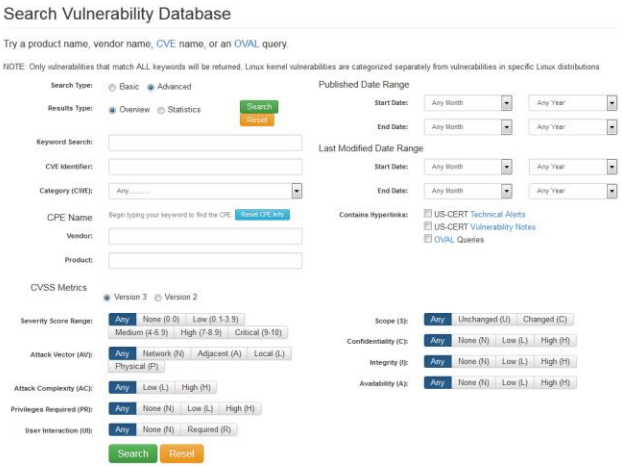

Fig. 2. Advanced Search Option of NVD Data

The NVD website provides a user interface for looking up information about all the CVEs and their corresponding information stored in its data set. In the base search form, there is only an option to perform keyword search. The NVD application looks for whatever the user has inputted in its database and shows results for entries with the search string in them. NVD also provides an advanced search option. In this web form, it allows users to search for any keyword, CWE category, CVE-Id as well as a large number of different options such as date-range (month and year) and CVSS scores. Fig. 2 shows an image of the web form for the advanced search page. In both the basic and advanced search options, there is an option for keyword search in this search function. The user can type in search phrases like “buffer overflow”, “X 509”, “Android”, or “2.3”, to look up vulnerabilities that match these keywords. In the advanced search page, a user can choose a CWE category from a

drop-down list and search for all the CVEs that have been categorized under that CWE. There are also a number of fine grained search options related to the different fields of a CVSS 2.0 or 3.0. When the user chooses these additional search criteria, the search functionality queries its database and returns the number of CVEs that met the criteria. If there are hits, the CVEs are listed as shown in Fig. 3.

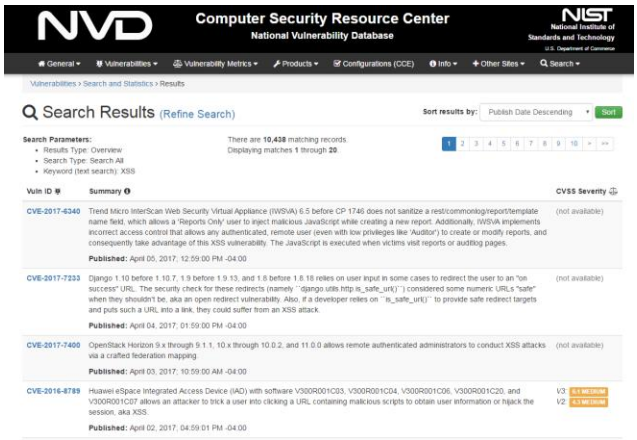

Fig. 3. Search Result Page of NVD

A. Combinatorial Input Model for Search Strings

One of our objectives for this study has been to systematically discover if there are search strings that may result in unexpected behavior from the NVD system. For any web application, coming up with an effective set of search strings to test the search functionality (e.g., keywords search) of the system is one of the common challenges for any test designer.

Instead of focusing only on likely keywords that users may use in a search, we approached the problem with a goal of creating keywords that are a combination of expected inputs such as simple strings and potentially unexpected symbols. Our hypothesis is that how the system responds to such rarely used search strings may not have thoroughly been tested for many web applications.

Parameter Name	Parameter Type	Parameter Value
First	Enum	[(,{,[,quote,sp,NUL,~,!,`,2]
Second	Enum	[sp,string,NUL,.]
Third	Enum	[and,or,amp,pipe,sp,NUL,-,/,backslash,3]
Fourth	Enum	[sp,string,NUL,.]
Fifth	Enum	],},],quote,sp,NUL,~,%,',4]

Fig. 4. Input Model for Generating Test Search Strings

We have taken the combinatorial approach to synthesize the search strings. Fig. 4 shows the five parameters, whose values are combined to create the test strings. Each parameter is comprised of a set of strings or special characters. There are 10, 4, 10, 4, and 10 enumerated values in the respective five parameters. All possible combinations of these values create 16,000 possible test strings. In addition to the generic term "string" to represent any string, there are two special strings: "and" and "or". The term "sp" represents space and "NUL" represents an empty string. The use of "NUL" allows us to synthesize strings that can have different special characters at different positions of the synthesized strings including at the beginning and at the end of the strings. Other special characters include different types of left (*(,{,[*) and right (*),},]*) brackets, single (`, ') and double (") quotes, dot (.), ampersand (&), pipe (|), exclamation sign (*!*), hyphen (-), percent sign (*%*), slash (*/*), backslash (\), and tilde (~). These special characters are not chosen completely randomly. Since NVD allows searching for any string in CVE descriptions and other associated information within the vulnerability database (full-text search), it is conceivable that some users may construct search strings with special characters that they are looking for within the descriptions.

B. Test Set Generation

Using the input model described above, 2-way, 3-way, and 4-way covering arrays were constructed using the ACTS tool. Test set sizes and failures are shown in Table I.

TABLE I. *t*-WAY TESTS AND RESULTS, *t*=2,3,4

t	Number of tests	Number of failed tests	% of failures
2	100	12	12.0
3	999	129	12.9
4	3125	473	15.1

TABLE II. RANDOM TESTS AND RESULTS

Test set	Number of Tests	Number of failed tests	% of failures
random 1	100	13	13.0
random 2	100	17	17 .0
random 3	100	13	13 .0
random 4	100	12	12 .0
random 5	100	7	07 .0
random 6	100	7	07 .0
random 7	100	7	07 .0
random 8	100	11	11 .0
random 9	100	12	12 .0
random 10	100	12	12.0

Because random or "fuzz" testing is often used in database testing, we generated random test sets of the same size as each of the *t*-way test sets, with results as shown in Table II for $t = 2$. Tests were generated by randomly selecting a value from each of the five factors detailed above. Results varied significantly and in three cases of the 10 runs, more failures were found in the random tests than with 2-way covering arrays. This occurred because with random generation, multiple occurrences of a fault-triggering combination would appear in some test sets more than others. Following the test runs, we used the fault location tool described in [34] to locate combinations that occurred

in failing tests that were not also in passing tests, as described in the next section.

V. Results and Discussions

The NVD is a heavily used database, averaging approximately 7.3 million accesses per month. It was tested extensively in development, and has been in continuous use, in some form, since 1998. Its usage profile is not unlike many other large, widely used information systems. While faults have been discovered occasionally, the system continues to perform adequately for the users who rely on it. Whenever faults have been found, they have been repaired quickly and have not disrupted service. The NVD group implemented a new version of full-text search in Spring 2016 and became aware of some new issues. They fixed some of the problems with special characters. Faced with the classic SE constraints of inadequate time and resources as well as tools and techniques to easily identify all failure scenarios with the new implementation, they approached us for a systematic and thorough testing of their search functionality.

The faults identified in this paper show that even long-term operation does not guarantee eventual discovery of *all* failures. Moreover, any new feature implementation may cause a number of new failures, which is unlikely to be discovered by a traditionally designed test suite. The failure-triggering combinations found in this study are clearly "corner cases", very unusual combinations of character strings that are unlikely to occur in practical use. From a $4^2 10^3$ input configuration, we identified 49 input string combinations that result in non-timeout related failures, or roughly 0.3% of the 16,000 possible combinations in the input space as modeled. It is notable that all of these are 2-way combinations containing at least one special character.

Fault-triggering combinations can be determined using simple heuristics described in [34]. More sophisticated methods exist for fault location, e.g. [32][33], but the simple heuristics below are quick and easy to apply for this test problem. For a deterministic system, in which a given set of input values always produces the same result independent of the order of variable values, let P = {combinations in passing tests} and F = {combinations in failing tests}. The following rules were applied:

- *Elimination*: For a deterministic system, $F \setminus P$ must contain the fault-triggering combinations because if any of those in were in P, then the test would have failed.
- *Interaction level lower bound*: If all t-way tests pass, then clearly a t-way or lower strength combination did not cause the failure.
- *Interaction continuity*: For each level of t, we compute $S_t = F_t \setminus P_t$, the suspicious t-way combinations that may have triggered a failure. Because t-way tests cover all combinations of t-way or lower strength, a combination that triggered the failure in F_t must also occur in F_{t+1}, F_{t+2}, etc. So we remove any combination in S_t from S_k for any $k > t$.

Initially, $F \setminus P$, combinations in failed tests not also in passed tests, were as shown in Table III. These sets were reduced by testing each individually, resulting in 49 2-way combinations, 144 3-way combinations, and 373 4-way combinations that all triggered failures. However, a lower strength combination that triggers a failure would also produce a failure if it is contained in any higher strength combination. For example, if the 2-way combination &% triggers a failure, it will also do so in a 3-way combination &%{. Therefore, suspect 3-way combinations were removed from the 4-way suspect set and suspect 2-way combinations were removed from the 3-way and 4-way sets. As shown in Table 1, it was then possible to conclude that only 2-way combinations were responsible for all failures discovered. The complete set of failure-triggering combinations is shown in Table IV.

Table III. Interaction identification

	2-way	3-way	4-way
Initial	49	144	373
removing (t-1)-way		0	124
removing (t-2)-way			0

It is important to note that if the test goal is strong assurance that all faults have been discovered, then 3-way and 4-way testing are necessary, even though they do not discover any additional failing combinations. Without running these stronger interaction tests, we would not have been able to conclude that the 2-way combinations likely represented the complete set of faults for this input model. Any reasonable testing scheme will require that we continue testing as long as errors are being discovered. High strength covering arrays provide a stopping criterion. If no new failures are discovered after increasing t-way coverage to (t+2)-way, it is unlikely that any new faults will be found.

Table IV. Failure triggering combinations

&%	.~	or~	|4	~3
&'	/~	str&	|str	~4
&.	2&	str|	|}	~\
&4	2|	str~	|~	~and
&str	2~	{&	~%	~or
&}	3~	{|	~&	~str
&~	\~	{~	~'	~|
-~	`&	|%	~-	~}
.&	`|	|'	~.	~~
.|	`~	|.	~/	

Text searches are among the most common tasks in information systems. Although the test procedure described in this paper addresses only this narrow problem, it is designed to be usable across the broad range of systems that require text search.

Covering arrays vs. random tests: Because *fuzz testing* is commonly used in many test situations, we compared the combinations covered by *t*-way arrays with coverage for an equal number of randomly generated tests. Fig. 5 shows a representative example for a random test set of the same size as a 3-way test set developed for the input model described in Sect. IV. The area under a curve represents the total combination coverage [35] for a given level of *t*, and the right-hand Y intercept represents the minimal coverage. For example, if variables are binary and there are one or more 2-way combinations where only 00 and 10 are covered (out of 00, 01, 10, 11), then the minimal coverage is 50%. The example test set in Fig. 5 shows approximately 95% 2-way, 84% 3-way, and 42% 4-way coverage. All 2-way combinations have at least 90% of settings covered, 3-way at least 65%, and 4-way at least 30% coverage.

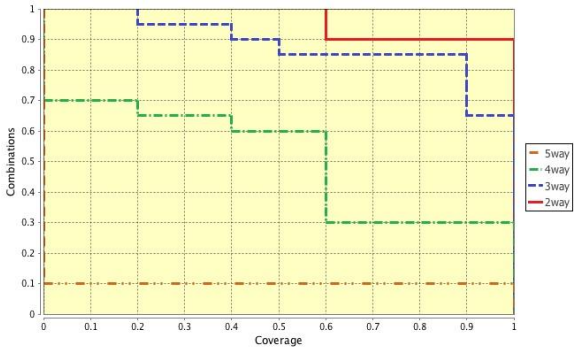

Fig. 5. Combinatorial coverage of random tests

Now consider what these coverage levels mean for assurance. With roughly 95% of 2-way combinations covered, we could expect to detect 46 or 47 of the 49 faults. So the fault detection capability of random tests, in this case, compares relatively well with covering arrays - *if we are not seeking high assurance*. Random testing falls short in two aspects for high assurance: 1) inadequate combination coverage for fault detection; and 2) inadequate coverage for a stopping criterion. For safety or mission-critical systems, finding only 95% of faults is unacceptable. Moreover, we would have no way of estimating the degree to which faults have been discovered without extending testing until the relevant input space has been covered.

Fuzz testing or other random test generation can be an efficient and appropriate means of fault discovery, but sound engineering requires a defensible method for measuring test thoroughness. Structural coverage metrics provide one set of reasonable measures - full branch or condition coverage indicates a degree to which executable code has been exercised. Measuring combinatorial coverage of tests can provide a complementary measure to structural coverage, because it shows what proportion of the input space has been included in tests. Any combination not covered by the test set is to some degree unknown territory - even with full structural coverage we do not know what the code will do with a particular combination of inputs. Similarly, extended use of a system does not guarantee that some inputs will not produce a failure, as shown by the NVD testing described in this paper. Using covering arrays makes it easy to check system response to rare inputs, to a degree that is unlikely and difficult to achieve with conventional test methods or through continuous use for many years.

VI. Future Work

Database-backed web applications usually also allow users to select advanced search options to select a subset of entries from the database that match multiple criteria. Our case study, NVD, is no exception. There is an advanced search option users can choose to narrow down their search results using many different criteria. Fig. 2. Shows an image of the NVD advanced search page. In addition to providing support for the keyword search, users can search for a specific CVE-Identifier such as CVE-2016-1234. There is option to select for a particular CWE category (e.g., CWE-94: Code Injection). Users can also choose a specific vendor or product to search for vulnerabilities associated with that vendor or product. There are two sets of date-ranges: published-date and last-modified-date that users can use to narrow their search. Additionally, there is a way to select options for different factors that define the CVSS scores of the known vulnerabilities stored in the database.

Like the keywords search, *CVE-Identifier*, *vendor*, and *product* search option allows the use of any string in the search fields. The other options are drop-down lists that users will have to choose an option from. There are 106 different CWE categories to select from in the NVD advanced search page. Similarly for CVSS version 2, users can choose from multiple options for *Severity Score Range* (SSR) of *Any*, *Low* (0-3), *Medium* (4-6), *High and Medium (4-10), and High (7-10)*. For *Attack Vector* of CVSS 2, there are options of *Any*, *Network* (N), *Adjacent* (A), and *Local* (L). There are other options available for selection for each of the other components of CVSS, which is comprised of *Access Complexity* (AC), *Authentication* (Au), *Confidentiality* (C), *Integrity* (I) and *Availability* (A). Even if we leave out the free-string search options for keywords, CVE Identifier, vendor, and product names, there are 1.45×10^{16} combinations of values for exhaustively testing the advanced search page of NVD application.

Since our goal is to look for search values and options that may lead to unexpected behavior or system failure, we

designed another input model with a selected subset of the parameter values for each of the search options. Fig. 6 shows the model we used to synthesize search strings to query the NVD database. In addition to utilizing the expected values for each of the advanced search options, we added an unexpected value such as "off" or "X".

Parameter Name	Parameter Type	Parameter Value
results_type	Enum	[overview,statistics,off]
cwe_id	Enum	[CWE-20,CWE-119,CWE-89,CWE-off]
pub_date_start_month	Enum	[-1,0,5,off]
pub_date_start_year	Enum	[1990,2000,2004,off]
publish_date_end_month	Enum	[6,11,12,off]
pub_date_end_year	Enum	[2007,2016,2020,off]
cvss_version2_severity	Enum	[LOW,MEDIUM,HIGH,OFF]
AV	Enum	[N,A,L,X]
AC	Enum	[L,M,H,X]
Au	Enum	[N,S,M,X]
C	Enum	[N,P,C,X]
I	Enum	[N,P,C,X]
A	Enum	[N,P,C,X]

Fig. 6. Input Model for Advanced Search Options

NVD allows direct querying of their database through the construction of search URLs. The designers left this option open for allowing programmatic search of different aspects of the information stored in the database. We utilized this feature to synthesize web search URLs combining parameter values shown in Fig. 6 and tested the NVD search engine responses. In our preliminary results, a large fraction of test cases resulted in a "Server Error" response. For example, 30 out of 33 test cases from 2-way and 767 out of 820 test cases from 4-way covering array produced an error response. The NVD developers indicate that these server errors are not necessarily bugs; rather a non-descriptive response to the end-user while processing invalid input.

It does appear that most of the unexpected parameter values result in server error response. However, not all unexpected values resulted in the error response. For example, a "-1" for *starting month* parameter results in a regular response from the application. Clearly there are some anomalies in how unexpected or invalid inputs are treated by the application. As future work, we plan to more deeply investigate the resiliency of NVD system against unexpected parameter values for advanced search options. We also plan to research the coverage we can gain from applying combinatorial testing approach over the 'Advanced Search's parameter input space. It would be interesting and useful to determine the combination factors that can cause 'Advanced Search' to fail. Initial test results have already revealed that certain valid CWE-category search can also cause failures while combined with other valid parameter values, which is something the application developers did not anticipate.

VII. Conclusions

We investigated the application of combinatorial testing to string text searches in the US National Vulnerability Database, a system that is accessed more than 70 million times a year. The current software build is operational 24 hours a day. Our testing and analysis revealed 49 inputs that produced server errors in the current build. These inputs were 2-way combinations of special characters and strings, and test cases built from 2-way through 4-way covering arrays demonstrated that no other combinations beyond these 49 resulted in the server error response. This result demonstrates the effectiveness of combinatorial methods for detecting and determining the full extent of rare faults.

The test procedure described in this paper addresses a specific test problem. It can be applied with little or no change to many systems that incorporate text searches. Text search is an essential component in systems within nearly all industries, and some are safety or mission-critical. Applying test methods such as those described in this paper can help to remove rare faults that could result in significant failures in operation.

Equally important, the methods described here provide a defensible criterion for test completion. Because covering arrays include all *t*-way factor combinations, we can show that the entire input space has been covered up to whatever *t*-way combinations are used. In contrast, "fuzz testing" or other conventional methods do not include measures of the input space that has been tested, and often rely on a "more is better" heuristic without an ability to measure completeness. Using covering arrays, or measuring combinatorial coverage of random tests, provides a sound test engineering method with defensible, quantitative measures of test completeness.

Acknowledgements

The authors would like to thank the NIST NVD group at for their assistance with this work. This study was supported by NIST ITL Grant 70NANB17H035.

Disclaimer: *Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose*

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," ACM Computing Surveys, vol. 29, no. 4, pp. 366–427, 1997.

[2] M. Heusser, "6 software development lessons from healthcare.gov's failed launch," CIO, November 2013.

[3] D. Doherty, "Team obama never finished testing healthcare.gov before launching it," CBS News, November 2013.

[4] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial software testing," Computer, vol. 42, no. 8, pp. 94–96, Aug 2009.

[5] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and R. Kuhn, "An empirical comparison of combinatorial and random testing," 2014 IEEE Seventh

Intl Conf on Software Testing, Verification and Validation Workshops, March 2014, pp. 68–77.
[6] K. Haller, "The test data challenge for database-driven applications," Third Intl Workshop on Testing Database Systems, ACM, 2010, pp. 6:1– 6:6.
[7] M. J. Su´arez-Cabal and J. Tuya, "Using an sql coverage measurement for testing database applica-tions," SIGSOFT Softw. Eng. Notes, vol. 29, no. 6, pp. 253–262,2004.http://doi.acm.org/10.1145/1041685.1029929
[8] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in Proceedings of the 2007 Intl Symp on Software Testing and Analysis, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 151–162.
[9] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," IEEE/ACM Intl Conf on Automated Software Eng., New York, NY, USA: ACM, 2010, pp. 289–292.
[10] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in Proc. of ICSE Future of Software Engineering (FOSE), 2007, pp. 85–103.
[11] D. Willmor and S. M. Embury, "An intensional approach to the specification of test cases for database applications," 28th Intl Conf on Software Engineering, New York, NY, USA: ACM, 2006, pp. 102–111
[12] NIST. (2007) National vulnerability database (nvd). [Online]. Available: https://nvd.nist.gov/
[13] MITRE/CVE. (2003) https://cve.mitre.org
[14] NIST/ CVSS. (2012) https://nvd.nist.gov/cvss.cfm
[15] Tsumura, K., et al., April. Pairwise coverage-based testing with selected elements in a query for database applications. *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth Intl Conf on* (pp. 92-101). IEEE.
[16] Bozic, J., Simos, D.E. and Wotawa, F., 2014, May. Attack pattern-based combinatorial testing. *9th Intl Wrkshp on Automation of Software Test* (pp. 1-7). ACM.
[17] Garn, B., Kapsalis, I., Simos, D.E. and Winkler, S., On the applicability of combinatorial testing to web application security testing: a case study. *2014 Workshp Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing* (pp. 16-21).
[18] Bozic, J., Garn, B., Simos, D.E. and Wotawa, F., April. Evaluation of the IPO-family algorithms for test case generation in web security testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth Intl Conf on* (pp. 1-10). IEEE.
[19] Ratliff, Z.B., Kuhn, D.R., Kacker, R.N., Lei, Y. and Trivedi, K.S., The Relationship between Software Bug Type and Number of Factors Involved in Failures. In *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE Intl Symp on* (pp. 119-124). IEEE.
[20] K. Haller, "The test data challenge for database-driven applications," Third Intl Workshop on Testing DatabaseSystems, ser. DBTest '10. New York, NY, USA: ACM, 2010, pp. 6:1– 6:6.
[21] Ghandehari, L.S., Czerwonka, J., Lei, Y., Shafiee, S., Kacker, R. and Kuhn, R., 2014, March. An empirical comparison of combinatorial and random testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh Intl Conf on* (pp. 68-77). IEEE.
[22] Schroeder, P. J., Bolaki, P., & Gopu, V. (2004, August). Comparing the fault detection effectiveness of n-way and random test suites. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 Intl Symp on* (pp. 49-59). IEEE.
[23] Ellims, M., Ince, D. and Petre, M., 2008, September. The effectiveness of t-way test data generation. In *Intl Conf on Computer Safety, Reliability, and Security* (pp. 16-29). Springer Berlin Heidelberg.
[24] Vilkomir, S., Starov, O. and Bhambroo, R., 2013, March. Evaluation of t-wise approach for testing logical expressions in software. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth Intl Conf on* (pp. 249-256). IEEE.
[25] Ballance, W.A., Vilkomir, S. and Jenkins, W., April. Effectiveness of pair-wise testing for software with boolean inputs. *Software Testing, Verification and Validation, 2012 IEEE Fifth Intl Conf* (pp. 580-586)
[26] Kobayashi, N., Tsuchiya, T. and Kikuno, T., 2001, July. Applicability of non-specification-based approaches to logic testing for software. In *Dependable Systems and Networks, 2001. DSN 2001.* (pp. 337-346). IEEE.
[27] Bell, K.Z. and Vouk, M.A., 2005, December. On effectiveness of pairwise methodology for testing network-centric software. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd Intl Conf on* (pp. 221-235). IEEE.
[28] Bryce, R.C. and Colbourn, C.J., 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints *Inf.Software Tech.*, *48*(10), pp.960-970.
[29] Ghandehari, L.S., Czerwonka, J., Lei, Y., Shafiee, S., Kacker, R. and Kuhn, R., An empirical comparison of combinatorial and random testing. *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh Intl Conf on* (pp. 68-77). IEEE.
[30] Bartholomew, R. (2013, May). An industry proof-of-concept demonstration of automated combinatorial test. In *Automation of Software Test (AST), 2013 8th Intl Workshop on* (pp. 118-124). IEEE.
[31] Borazjany, Mehra N., et al. "An input space modeling methodology for combinatorial testing." *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth Intl Conf on*. IEEE, 2013.
[32] Colbourn, C. J., & McClary, D. W. (2008). Locating and detecting arrays for interaction faults. *Journal of combinatorial optimization*, *15*(1), 17-48.
[33] Wang, Z., Xu, B., Chen, L., & Xu, L. (2010, July). Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th Intl Conf on* (pp. 495-502). IEEE.
[34] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010). SP 800-142. Practical Combinatorial Testing.
[35] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2016). Measuring and specifying combinatorial coverage of test input configurations. *Innovations in Systems and Software Engineering*, *12*(4), 249-261.