An Empirical Study of Prioritizing JavaScript Engine Crashes via Machine Learning

Sunnyeo Park
KAIST

Dohyeok Kim
KAIST

Sooel Son
KAIST

ABSTRACT

The early discovery of security bugs in JavaScript (JS) engines is crucial for protecting Internet users from adversaries abusing zero-day vulnerabilities. Browser vendors, bug bounty hunters, and security researchers have been eager to find such security bugs by leveraging state-of-the-art fuzzers as well as their domain expertise. They report a bug when observing a crash after executing their JS test since a crash is an early indicator of a potential bug. However, it is difficult to identify whether such a crash indeed invokes security bugs in JS engines. Thus, unskilled bug reporters are unable to assess the security severity of their new bugs with JS engine crashes. Today, this classification of a reported security bug is completely manual, depending on the verdicts from JS engine vendors.

We investigated the feasibility of applying various machine learning classifiers to determine whether an observed crash triggers a security bug. We designed and implemented CRScope, which classifies security and non-security bugs from given crash-dump files. Our experimental results on 766 crash instances demonstrate that CRScope achieved 0.85, 0.89, and 0.93 Area Under Curve (AUC) for Chakra, V8, and SpiderMonkey crashes, respectively. CRScope also achieved 0.84, 0.89, and 0.95 precision for Chakra, V8, and SpiderMonkey crashes, respectively. This outperforms the previous study and existing tools including *Exploitable* and *AddressSanitizer*.

CRScope is capable of learning domain-specific expertise from the past verdicts on reported bugs and automatically classifying JS engine security bugs, which helps improve the scalable classification of security bugs.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**; Browser security; • **Computing methodologies → Machine learning**.

KEYWORDS

Crash analysis; Machine learning; Security bugs; JavaScript; Browser security

ACM Reference Format:
Sunnyeo Park, Dohyeok Kim, and Sooel Son. 2019. An Empirical Study of Prioritizing JavaScript Engine Crashes via Machine Learning. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3321705.3329840

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329840

1 INTRODUCTION

A security bug in a popular web browser imposes a critical security threat, affecting billions of Internet users [57]. A motivated adversary who exploits zero- or one-day security vulnerabilities has conducted exfiltrating sensitive information [15, 45], performing remote code execution [11, 12, 44], and bypassing sandbox policies [13, 16, 43]. Unfortunately, today's browsers are not safe from zero- and one-day adversaries. According to 2018 statistics from the Kaspersky Lab, web browsers are the second most frequently exploited applications, accounting for one-fourth of all observed attacks [33].

Among the components of web browsers, the JavaScript (JS) engine is the core component most frequently targeted by adversaries. The majority of users enable JS support in their browsers because 95.2% of all websites are in JavaScript [62]. The Turing-complete nature of the JavaScript language also allows adversaries to compose attack code to trigger unexpected or undefined behaviors. Furthermore, JS engines have become increasingly large and complex as their browsers support more emerging HTML5 features. Security bugs in such a complex and dominant application are certainly attractive targets, which adversaries seek to exploit.

Thus, it is imperative for browser vendors to find these security bugs as early as possible before adversaries abuse them. Today, popular browser vendors depend on tech-savvy users, bug bounty hunters [18, 38, 42], and security researchers [27, 61] to report these security bugs. They also use state-of-the-art fuzzers and cloud computing resources to find defects [20, 53].

Popular browser vendors have managed Bug Tracking Systems (BTSs) to track reported bugs and reward bug reporters. Because the prompt patch of a reported security bug is important, vendors ask bug reporters to label whether their report is *security-related* [17, 35, 41]. Once a security bug is reported, a developer is assigned to determine whether it is indeed a security bug. This procedure requires domain expertise in the target browser and its engine, thus making the procedure completely manual. If a bug reporter submits a false positive, the engineering cost of a domain expert checking the reported bug will be wasted. Even worse, a bug reporter may post a security bug as a non-security bug, postponing the proper patch and providing an opportunity for an attacker to exploit the posted bug. Therefore, engine developers and bug reporters are in dire need of an automated oracle that classifies security bugs.

Previous studies in the field have focused on pinpointing bug locations by analyzing crashes [5, 64, 65], determining the exploitability of crashes [23, 60, 66], and performing natural language processing of bug reports [2, 9, 31]. Tripathi *et al.* proposed training

a binary classifier to determine whether or not a given crash due to bugs is exploitable. Their tool, called *Exniffer*, approximates the oracle which determines the exploitability of a bug by analyzing its associated crash [60]. Off-the-shelf tools also exist, including *Exploitable* and *AddressSanitizer*, which determine the exploitability of a bug when it causes a target binary to crash [8, 56]. However, no previous study has explored classifying security bugs in JS engines by analyzing their crashes. In this paper, we demonstrate that *Exploitable* and *AddressSanitizer* are unfit for classifying security bugs and that *Exniffer* does not achieve high accuracy or precision for classifying security bugs in JS engines.

Contributions. We design and implement a tool, named CRash Scope (CRScope). Given a JS engine crash, CRScope classifies whether a security bug causes the crash. Consider a bug reporter with a new JS test that causes a target engine to crash. CRScope provides a verdict on whether this JS test has triggered a security or non-security bug.

CRScope leverages a machine learning classifier trained on the past verdicts of reported bugs causing the crashes of JS engines. To train the classifier, we collected 165 JS PoCs triggering security bugs, and 174 JS tests triggering non-security bugs from Chrome, Mozilla, and Internet Explorer/Edge BTSs as well as GitHub repositories. For each collected JS test, we compiled a JS engine binary, which contained a past bug corresponding to the JS test. From the resulting 727 JS engine binaries, we prepared 766 crash instances on which the classifier model of CRScope was trained and validated. To the best of our knowledge, no previous study has built a ground truth dataset for JS engine security bug classification. To support open science and invite further research, we release our ground truth dataset and CRScope.

We evaluated six different classifiers of CRScope, including the Random Forest Classifier (RFC), Decision Tree Classifier (DTC), Multinomial Naive Bayes (MNB), Logistic Regression (LR), Linear Support Vector Classification (SVC), and Multi-layer Perceptron Classifier (MLP) in scikit-learn [4]. For each model trained on the crashes of Chakra, V8, and SpiderMonkey, CRScope achieved Area Under Curves (AUCs) of 0.85, 0.89, and 0.93, respectively. We demonstrated that CRScope outperforms *Exniffer*, the previous study that addressed the classification of crash exploitability.

Prioritizing JS engine bugs is important in making sure the security bugs get patched as soon as possible. Identifying security bugs in the first place requires domain-specific knowledge, which often causes false positive reports and hinders the scalable classification of security bugs. No prior tool, however, has been able to classify security bugs by analyzing JS engine crashes. CRScope leverages a machine learning classifier, trained on the past verdicts made by domain experts. CRScope is a new tool that can help bug reporters and developers automatically classify security bugs, regardless of whether they have domain expertise.

In summary, our contributions are as follows:

- We demonstrate that the current off-the-shelf tools, *Exploitable* and *AddressSanitizer* are unfit for classifying security bugs in JS engines.
- We build the first well-labeled dataset containing 165 security and 174 non-security JS engine bugs with their JS test code as well as their crashes. We release our dataset to support further research at https://github.com/WSP-LAB/CRScope.
- We design, implement, and evaluate CRScope, the first tool to classify JS engine security bugs via machine learning, achieving AUCs of 0.85, 0.89, and 0.93 for Chakra, V8, and SpiderMonkey crashes, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Security and Non-security Bugs

A JS engine is an interpreter that executes a given JS code snippet. All popular browser vendors have implemented and managed their own JS engine: V8 for Google Chrome [22], JavaScriptCore for Apple Safari [63], SpiderMonkey for Mozilla Firefox [48], and Chakra for Microsoft Internet Explorer/Edge [37]. These JS engines have increasingly become targets of adversaries abusing their vulnerabilities. Vulture showed that the majority of vulnerability fixes in Mozilla Firefox are due to JS engine components [51]. Considering the prevailing usage of JS code on the Internet and the wide-adoption of browsers, zero-day or one-day vulnerabilities of popular JS engines pose a critical threat. Adversaries often exploit these vulnerabilities by luring victims to visit the webpage with attack JS code, resulting in remote code execution.

These JS engines are lucrative targets to bug bounty hunters. Google, Mozilla, and Microsoft offer rewards of $15,000 or even more for the accurate reporting of these security bugs [18, 38, 42]. Furthermore, Zerodium pays ten times more for these security bugs [67]. The monetary motives and research interests in finding security bugs have spurred browser developers, tech-savvy users, and intrigued researchers on to report 161 Chrome CVEs in 2018 alone [7].

To manage and track such reported bugs, each popular browser vendor has developed a Bug Tracking System (BTS) [19, 36, 46]. These BTSs demand a specific bug report when a reporter files a new bug. They also expect the reporter to label the found bug according to the guidelines defined in each BTS. For instance, the Chrome BTS requires a bug reporter to specify (1) which component a bug belongs to, (2) whether the bug causes any crashes, and (3) whether the bug is security-related.

This security label is important for browser vendors to prioritize the reported security bug from other non-security bugs. It is expected that software vendors will patch any security-related bugs within 90-days prior to public disclosure; this security label is the sole indicator for prioritizing bug patches. However, it is possible for a bug reporter to mislabel a security bug. Thus, browser vendors should validate whether the filed bug is indeed a security-related bug.

Today, this procedure is completely manual and unclear, and it requires domain expertise, thus making the entire procedure non-scalable and even erroneous. For instance, there were two Chrome bug reports [10, 14] that were filed as non-security bugs. However, it took ten months and five months, respectively to find that they were, in fact, security related. CVE-2017-5132 was later assigned to one of the bugs, which gave a significant time window to exploit this publicly available vulnerability. If the bug reporters had filed the reports as security bugs with some confidence, these vulnerabilities would have been patched within several days.

To make the prioritizing procedure prompt and scalable, we propose CRScope, which classifies a given security bug by leveraging

machine learning models trained on past verdicts. Our goal is to train a model capable of classifying security bugs without requiring domain-specific expertise.

2.2 Crash-dump

Fuzzing is a common and prevailing technique for finding JS engine bugs [20, 27, 53, 61]. It intentionally generates invalid or unexpected input to a program, then monitors whether or not the target program crashes. When a bug triggers a target JS engine to crash, the underlying OS of the JS engine creates a file, termed a *crash-dump*.

A crash-dump, also referred to as a *core-dump*, retains a recorded process state at the time of its termination. It contains a snapshot of main memory and CPU registers, including a program counter and stack pointer. It also has a process terminal signal such as SIGSEGV, SIGABRT, SIGILL, SIGFPE, or SIGQUIT. Due to its context-rich information, a core-dump file has been used for debugging a bug and triaging its cause. Users do not require any purpose-built tool to obtain a crash-dump, nor does it impose any performance overheads.

The idea of leveraging a crash-dump to predict whether a bug causing a crash is *exploitable* was explored. *Exploitable* and *Address-Sanitizer* are public tools that analyze a crash-dump and determine whether the bug causing the crash is exploitable. A naive approach for tech-savvy users without domain expertise is to use these off-the-shelf tools that infer the exploitability of a bug causing a JS engine to crash. *Exploitable* [8] is a gdb extension that inspects the state of a Linux application that has crashed; it then classifies a bug as to how difficult it might be for an attacker to exploit. *Address-Sanitizer* [56] is another tool that assigns a memory corruption label to a given crash. Mozilla uses this label to classify exploitable crash reports [47] and ClusterFuzz from Google leverages this label to rate the security severity of a found crash. When the label is Bad-cast, or Heap-use-after-free, the found crash is classified as a high severity bug [21]. Note that both *Exploitable* and *Address-Sanitizer* are designed to gauge the exploitability of a given crash, *not to classify whether a given crash is security-related*. We thus conducted experiments to measure the efficacy of the two tools in predicting security bugs. Section 3 and Section 6.1 describe in detail our experiments on 339 bugs and their 766 crashes.

In Section 6.1, we demonstrate that *Exploitable* achieved 0.48 precision, producing 85% false positives. If a browser vendor uses *Exploitable* to prioritize security bugs, the engineering cost of vetting false positives would be unnecessarily wasted. Also, *Address-Sanitizer* achieved 0.76 precision and 0.60 recall, producing an unacceptable number of false negatives. 190 crashes (42%) triggered security bugs that *AddressSanitizer* was unable to detect. These experimental results demonstrate that neither tool is fit to classify security bugs in JS engines.

To address the shortcomings of these off-the-shelf tools, we design and implement CRScope leveraging machine learning models to classify security bugs in JS engines. Anyone, including technically novice users who are motivated to find security bugs in JS engines, can use state-of-the-art fuzzers, find a JS test causing a JS engine crash, and claim the security bug with confidence by leveraging CRScope. Engine vendors also benefit from automatically classifying reported bugs to filter spurious bug reports.

Table 1: Ground truth data for JS engines

JS engines	PoC code			Crash instances		
	Security	Non-Security	All	Security	Non-Security	All
Chakra	69	35	104	126	53	179
V8	50	41	91	147	115	262
SpiderMonkey	46	98	144	95	230	325
All	165	174	339	368	398	766

3 DATASET

CRScope is a binary classifier that assigns either a "*security bug*" or "*non-security bug*" label to a given JS engine crash-dump. To train this classifier, we built the ground truth dataset, a collection of JS engine crash-dump files that were manually classified as security and non-security bugs. We collected these past classification verdicts from BTSs and GitHub repositories for V8, SpiderMonkey, and Chakra.

Collecting PoCs. For security bugs, we collected CVEs with PoC code reported between 2011 and 2018. From each Chakra release note enlisted in the GitHub repository, we collected patched CVEs [58]. From the Chrome BTS, we also collected bug reports, each of which was assigned with a CVE or rewarded from Google [19]. For Mozilla, we leveraged a list of known security vulnerabilities and collected CVEs of SpiderMonkey [49]. We then searched PoC code for the collected CVEs from patch commits, bug reports, and websites that have been archiving exploits [26, 55].

For non-security bugs, we collected test cases that did not have security-related labels. For each Chakra bug, we read its corresponding GitHub issue and selected crashing bugs triggering non-security bugs. In the case of V8 and SpiderMonkey, we collected bug reports with PoC code, but without assigned CVEs or rewards.

Building JS engine binaries. There is a remaining challenge. It is to find the correct version of a target JS engine for each PoC triggering a security or non-security bug. Our objective is to prepare the engine binary which generates a crash-dump at the time the bug is reported. Unfortunately, we observed that a large number of bug reports missed information on their target JS engine version where the bugs were found. For each bug report without an explicit version, when we knew its patch commit or patch date, we retrieved the JS engine repository for which the last commit was ahead of the patch date. When we could not even obtain the patch information, we restored the repository for which the latest commit date was ahead of the date when the report was filed. For instance, if the bug report was filed on January 15, 2019, we retrieved the latest repository whose commit occurred prior to January 15, 2019.

Given a labeled PoC code snippet, we prepared JS engine binaries to produce crash-dumps from which CRScope extracts features. Note that each JS engine supports both *ia32* and *x64* architectures except for Chakra; Chakra only supports *x64* architecture. Each JS engine also provides *debug* and *release* modes. Therefore, each test for a target JS engine can have up to four corresponding binaries.

Table 1 summarizes our ground truth dataset. We collected a total of 766 crash instances by running 339 PoCs on V8, SpiderMonkey, and Chakra engines for each architecture and each mode. 165 security bugs (69 in Chakra, 50 in V8, and 46 in SpiderMonkey

Table 2: List of features used in CRScope

Feature Name	Preprocessed	Feature Extraction
JS engine name	✗	LabelEncoder
Architecture type	✗	LabelEncoder
Compile mode	✗	LabelEncoder
Signal type	✗	LabelEncoder
Crash type	✗	LabelEncoder
Crashing instruction	✗	TfidfVectorizer, CountVectorizer
Crashing function	✓	TfidfVectorizer, CountVectorizer
Backtrace	✓	TfidfVectorizer, CountVectorizer

PoCs) caused 368 JS engine binaries to crash. The others are caused by 174 non-security bugs. Whereas the number of V8 PoCs is the smallest, the number of Chakra crash instances is smaller than the others. It is because the number of Chakra engine binaries is smaller than others due to not supporting the *ia32* architecture.

Difficulties in building the ground truth. *We emphasize that preparing the ground truth data for major JS engines is an arduous task.* There was no previous study or public project involving the classification of JS engine crashes. There are also a limited number of CVEs with publicly available PoC code, thus hindering the collection of training instances. Furthermore, to collect crashes from the binary when the report was filed, we carefully read every bug report and validated whether the corresponding PoC code worked.

We compiled 727 binary instances, which accounted for 1200GB (1.2TB). This task was also challenging because compiling old versions of JS engines often requires the usage of deprecated libraries and outdated OS supports. In some instances, building the old versions of JS engines required investigation and non-trivial engineering costs because the method of building each JS engine has changed over time. If a rolled back repository contained a compilation error, we had to manually fix this error by looking through commit logs. Compiling 727 binaries took a long time and required significant computing resources. We did not intend to miss any bugs with PoC code resulting in a target JS engine crash. Three graduate students invested nine months in identifying and validating 339 bugs and their bug reports from Chrome, Mozilla, and Chakra BTSs. To support open science and further research, we release the ground truth data and source code at https://github.com/WSP-LAB/CRScope.

4 METHODOLOGY

This section describes how we design and implement CRScope. Section 4.1 explains each feature that CRScope extracts from a crash-dump and their preprocessing. Section 4.2 describes six classification models that we evaluate to select a proper model for CRScope.

4.1 Extracting Features

Given a crash-dump file, CRScope extracts eight different feature types. Table 2 summarizes the extracted features. The first and second columns represent their types and whether CRScope preprocessed them, respectively. The last column indicates the method of vectorizing each feature. We applied three different vectorizing methods: *LabelEncoder*, *CountVectorizer*, and *TfidfVectorizer* [4].

LabelEncoder encodes a feature value into a number which varies from zero to the number of its unique instances minus one. For the feature of JS engine name, Chakra, V8, and SpiderMonkey are encoded into 0, 1, and 2, respectively.

CountVectorizer converts a feature consisting of various elements into the vector of element frequencies over the dataset. Note that *crashing instruction* feature has many vocabularies, e.g., opcode and operand. After conducting the *CountVectorizer* extraction, the crashing instruction feature becomes a sparse vector with the counts for each vocabulary contained in the crashing instruction.

TfidfVectorizer operates similar to *CountVectorizer* except for being able to filter out elements with frequencies that are too low or too high, by computing the Term Frequency - Inverse Document Frequency (TF-IDF) value for each element [30].

Before applying the *CountVectorizer* and *TfidfVectorizer* methods, we extracted *n-gram* tokens from each crashing instruction, *crashing function*, and *backtrace* while varying *n* from one to five. We encoded these tokens into feature vectors, thus capturing the relationships between operations, operands, and call sequences.

We applied *LabelEncoder* to encode the top five features in Table 2 and vectorized the bottom three features using *TfidfVectorizer* and *CountVectorizer*. All of the encoded vectors above are concatenated into one large vector for each crash instance. The lengths of the vectors generated using the Chakra, V8, and SpiderMonkey datasets are approximately 18,000, 24,000, and 30,000, respectively. However, significant features are only subsets of them, and an unnecessarily large size of vector dimension hinders the correct classification of crash instances [68].

We reduced the feature vector dimension by conducting two preprocessing steps. Hall *et al.* stated that a good feature subset should contain features uncorrelated with (not predictive of) each other [25], which guided our correlation-based feature selection process. First, we computed a Pearson correlation coefficient [3] for every feature pair. For each feature, we removed highly correlated features the coefficients of which were over 0.9 because these features are redundant in performing classifications [25]. We then selected 100 features after conducting *SelectKBest* from [4], which internally performs a chi-square test. It computes chi-squared statistics between each feature and class and eliminates features that are likely to be irrelevant for classification.

The followings describe the details of our extraction method for each feature type.

JS engine name. Each browser vendor could have subtle differences in the criteria for determining a security bug. We thus provide a target engine name to a model and let the model pick this feature or not while training. Its feature value is among Chakra, V8, and SpiderMonkey.

Architecture type and compile mode. A JS engine has up to four different binaries for the supporting architectures and compile modes (See Section 3). Since crash-dumps and other extracted features differ among these four different binaries, we feed a target model with *ia32* and *x64* for the architecture type as well as *release* and *debug* for the compile mode.

Signal type. When a process is abnormally terminated, it receives a terminal signal, then crashes. This terminal signal depends on the root cause of the crash. For instance, SIGSEGV is a segmentation violation signal which informs that a process attempted to access a

not-allowed memory region. SIGABRT is an abort signal indicating that the process is unable to run further because of asserting statements or other causes. The feature value is either SIGSEGV, SIGILL, SIGABRT, or SIGFPE.

Crash type. *Exploitable* is a gdb plugin that determines the exploitability of a crash-dump file. It also classifies a crash type. There exist 22 different crash types such as ReturnAv, BranchAv, DestAvNearNull, and BadInstruction. Each crash type represents an instance where access violations occurred or means that the current crashing instruction is illegal. Although we use *Exploitable*, it is trivial to compute this label by analyzing a crash-dump without it.

Crashing instruction. CRScope extracts the instruction at which the crash occurred. This instruction provides useful information about the crash cause. If a crashing instruction is a read or write operation, the main cause of the crash could be due to the reading or writing of an invalid memory, respectively.

An instruction consists of an opcode and its operands. CRScope extracts n-grams (from one to five) from this instruction. For example, from the mov eax, ebx instruction, CRScope extracts mov, eax, ebx, mov eax, eax ebx, and mov eax ebx. Each n-gram token becomes a feature element, the value of which is its frequency over the entire dataset. That is, CRScope extracts a vector for which the element corresponds to an n-gram token frequency for each of the *TfidfVectorizer* and *CountVectorizer* algorithms.

Crashing function. The function at which the crash occurred is the top frame of the stack trace. It is the actual function where the crash occurred. Schroter *et al.* found that 40% of bugs were fixed in these top functions [54], and Wu *et al.* located 50.6% of crashing faults by examining the top function [64]. We thus assume that a crashing function, the top of a stack trace, is closely correlated with crashes. We feed this feature into the CRScope classifiers. CRScope also preprocesses this feature by transforming a function name into a sequence of its namespace, class, and member function name, excluding arguments and templates. It then extracts n-gram (from one to five) tokens from this sequence. Our intention behind this scheme is to capture the semantic that appeared in word tokens of function names.

Backtrace. A backtrace (also called *stack trace*) is an ordered list of callees or active stack frames when the crash occurs. This stack trace is one of the most useful pieces of information when developers want to identify the root cause of the crash by tracking execution flows. In practice, many software vendors send these backtraces from client crashes to triage the problems. Schroter *et al.* demonstrated that bug reports with backtraces had got resolved significantly sooner than other bug reports [54]. The intuition behind collecting this feature is that when a crash occurs, any callee in the stack trace is highly correlated with security bugs. We expect that the function name itself contains a semantic of its internal logic, which hints at a model in performing the classification.

CRScope preprocesses the extracted backtrace features. For a given backtrace, CRScope only extracts the sequence of callees, each of which only contains a function name excluding other parts such as namespaces and classes. It then extracts *n*-gram (from one to five) tokens, each of which becomes a callee sequence with length *n*.

Table 3: Hyperparameters of each model explored via grid search

Models	Parameters	Values
MNB	alpha	0.01, 0.1, 1, 10, 100
DTC	max_features	sqrt, auto, log2, None
	splitter	best, random
	criterion	gini, entropy
	max_depth	None, 2, 5, 10, 20, 50, 100, 500, 1000
RFC	max_features	auto, sqrt, log2, None
	n_estimators	10, 50, 100
	warm_start	False, True
	criteriton	gini, entropy
	max_depth	2, 5, 10, 20, 50, 100
SVC	penalty	l1, l2
	loss	hinge, squared_hinge
	dual	True, False
	tol	1e-4, 1e-6, 1e-8, 1e-10, 1e-20
	C	1, 10, 100, 1000, 10000
	max_iter	100, 1000, 10000, 100000
	multi_class	ovr, crammer_singer
LR	penalty	l1, l2
	dual	True, False
	tol	1e-4, 1e-6, 1e-8, 1e-10
	C	1, 10, 100, 1000
	solver	liblinear, newton-cg, lbfgs, sag
	max_iter	100, 1000, 10000, 100000
	warm_start	False, True
MLP	hidden_layer_sizes	(100,), (100, 25,)
	activation	identity, logistic, tanh, relu
	solver	lbfgs, sgd, adam
	alpha	1e-4, 1e-2, 1, 100
	learning_rate	constant, invscaling, adaptive
	tol	1e-4, 1e-6, 1e-8, 1e-10, 1e-20
	epsilon	1e-1, 1e-2, 1e-3, 1e-5, 1e-10

4.2 Classification models

Using features extracted as described in Section 4.1, we evaluated Multinomial Naive Bayes (MNB), Decision Tree Classifier (DTC), Random Forest Classifier (RFC), Linear Support Vector Classification (SVC), Logistic Regression (LR), and Multi-layer Perceptron Classifier (MLP) machine learning models in the scikit-learn package [4]. Our evaluation goal is to find a model with the best performance for CRScope in classifying security bugs.

Each model requires different hyperparameters, thus making hyperparameter tuning imperative because they affect evaluation results. We thus conducted the grid search in [4] to find the best hyperparameters for the models referred to above. The employed grid search is an accuracy-guided exhaustive search technique, thus selecting the set of hyperparameters that produces the highest accuracy.

Table 3 shows hyperparameters that we explored by conducting the grid search. For each model, hyperparameters in the second column are determined by exhaustively searching over specified values in the third column. We used scikit-learn default values for other parameters. For the details of each hyperparameter, we

Table 4: Example of the grid search for DTC model

Parameters					Accuracy
(1)	(2)	(3)	(4)	(5)	
		...			...
log2	10	True	gini	2	0.719 ± 0.095
log2	10	True	gini	5	0.724 ± 0.060
log2	10	True	gini	10	0.737 ± 0.041
log2	10	True	gini	20	0.742 ± 0.123
log2	10	True	gini	50	0.77 ± 0.118
auto	10	True	gini	100	0.76 ± 0.0093
sqrt	10	True	gini	100	0.77 ± 0.077
log2	10	False	gini	100	0.783 ± 0.101
log2	**10**	**True**	**gini**	**100**	**0.788 ± 0.121**
log2	50	True	gini	100	0.77 ± 0.077
log2	100	True	gini	100	0.77 ± 0.077
log2	500	True	gini	100	0.765 ± 0.075
log2	1000	True	gini	100	0.765 ± 0.075
log2	5000	True	gini	100	0.765 ± 0.075
log2	10000	True	gini	100	0.765 ± 0.075
None	10	True	gini	100	0.742 ± 0.029
log2	10	True	gini	500	0.756 ± 0.085
log2	10	True	gini	1000	0.77 ± 0.056
log2	10	True	entropy	100	0.747 ± 0.068
		...			...

(1) max_features (2) n_estimators (3) warm_start (4) criterions (5) max_depth

refer the interested readers to the technical documentation for the scikit-learn package [4] for more details.

Table 4 illustrates an example of the grid search result conducted on a DTC model. The grid search generates all possible combinations for a given set of hyperparameter values and then calculates the accuracy of the model trained with the corresponding parameters. Each column in the first five columns represents a parameter, and the last column is the accuracy. Each row shows each parameter candidate. In Table 4, the candidate [$log2$, 10, $True$, $gini$, 100] in bold produces the best average accuracy. As such, each parameter of CRScope is determined by the grid search, producing the best performance.

5 EXPERIMENTAL DESIGN

We evaluated CRScope by answering three key questions:

- **RQ1**: How effective are *Exploitable* and *AddressSanitizer* in classifying security bugs from given crash-dumps?
- **RQ2**: Is CRScope effective in classifying security bugs from given crash-dumps? If so, how much more accurate is CRScope in comparison to the previous tools?
- **RQ3**: What feature types are most significant in classifying security bugs?

To answer these questions, we evaluated six different classifiers trained on features that CRScope extracts. The following sections explain our cross-validation methodology and the method for handling imbalanced training instances.

5.1 Cross-Validation

Our evaluation goal is to check whether a CRScope model can be trained on past bugs and their crash-dump files, and whether it is able to classify an unforeseen crash-dump. All browser vendors accept a security bug report only if the reported bug exists in the latest versions of their browsers or JS engines. Conservatively, a model should be capable of classifying a crash on the latest JS engines, which the model has not observed at all.

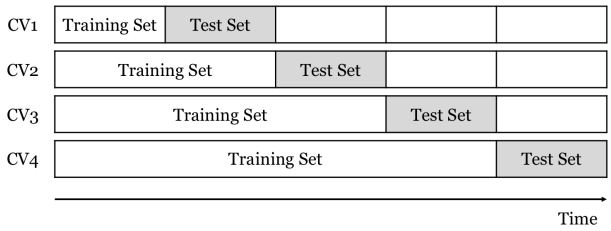


Figure 1: Four-fold time series cross-validation in CRScope

Cross-validation is a prevalent technique for evaluating a machine learning model. A standard n-fold cross-validation shuffles crash instances to create training and test sets so that crash instances on a later JS engine version will be used for predicting a past bug, which was reported ahead of the release of the JS engine. This is a common pitfall of the previous research on hardware-counter based malware detection via machine learning, as Zhou *et al.* pointed out [68]. That is, a standard n-fold cross-validation is not applicable for evaluating CRScope because our dataset is by nature a time series.

We designed our evaluation to check whether CRScope is able to classify future security bugs. We sorted all security and non-security bugs by their commit dates, which we used in their target binaries. We then arranged their crash instances in ascending order and divided these crashes into five bins so that each bin had the same number of crash instances. From those five bins, we conducted the 4-fold cross-validation. The first cross-validation used the first and second bins as the training and test sets, respectively. Its successive cross-validation used all bins from the previous test as the training set and used the next bin as the testing set, as shown in Figure 1. Specifically, with five bins $\{1, 2, 3, 4, 5\}$, we leveraged four cross-validation sets of $\{Training : 1 / Testing : 2\}$, $\{1, 2 / 3\}$, $\{1, 2, 3 / 4\}$, and $\{1, 2, 3, 4 / 5\}$.

5.2 Balancing instances

Balancing security-related and non-security crashes is vital to avoiding the classification bias towards a majority class [68]. When building and evaluating a model on imbalanced data such that the number of instances per each class is not equally distributed, classifier models are more likely to classify a new observation to the majority class because the probability of instances belonging to the majority class is significantly high.

Tripathi *et al.* evaluated Support Vector Machine (SVM) models in classifying exploitable crashes [60]. However, they used a total of 523 crashes with 166 exploitable and 357 non-exploitable samples. They trained and evaluated their model using this imbalanced dataset. Thus, their evaluation results were biased towards non-exploitable crashes.

We also have an imbalanced dataset. To address this problem, it is feasible to increase the frequency of the minority class, or to decrease the frequency of the majority class. To avoid missing any crash instances due to under-sampling, we performed over-sampling to balance security and non-security crashes, when

Table 5: Evaluation of *Exploitable*

Our verdict	*Exploitable*	Crash instances	
		Security	Not-security
Exploitable	exploitable	269 (73.10%)	**197 (49.50%)**
	probably-exploitable	40 (10.87%)	**140 (35.18%)**
Not-Exploitable	probably-not-exploitable	**15 (4.08%)**	52 (13.07%)
Ignored	unknown	44 (11.96%)	9 (2.26%)

preparing the cross-validation set. We applied the random oversampling algorithm [34], which duplicates random records from the minority class.

6 EXPERIMENTAL EVALUATION

6.1 Performance of Exploitable and ASan (RQ1)

Exploitable. *Exploitable* [8] is a gdb extension. It analyzes the current execution state of a target process when the gdb pauses the crashing process with a crash-dump. It then predicts the exploitability of this crash-dump by leveraging pre-defined heuristics. Each label becomes the sole factor in determining the exploitability of a target crash. This determination is classified as either exploitable, probably-exploitable, probably-not-exploitable, or unknown.

Table 5 describes the evaluation results of *Exploitable* on 339 PoCs and their 766 crashes. The second column represents a label that *Exploitable* predicts. The third and fourth columns represent the true labels of tested crashes. The first column shows our interpretation of the *Exploitable* reports. We ignored the unknown label for *Exploitable* because we were unable to make a determination with the information provided. Hence, we conservatively excluded these 53 crash instances when computing the precision, recall, and accuracy of *Exploitable*.

Approximately 84% of security crashes were labeled exploitable or probably-exploitable, which we considered to be security-related. That is, *Exploitable* achieved 0.95 recall for security bugs. However, the resulting accuracy was 0.51 and the precision was 0.48 due to false positives and false negatives, which are marked bold in the table; 85% of non-security bugs were classified as *exploitable* (false positives), and 4% of security bugs were classified as *not-exploitable* (false negatives). If a browser vendor used *Exploitable* to prioritize security bugs, the engineering cost of vetting these false positives would be wasted. Furthermore, there still exist 53 crashes that *Exploitable* is unable to classify.

AddressSanitizer. *AddressSanitizer* [56] is an open-source memory error detector from Google, which is designed to detect memory-related bugs such as use-after-free and buffer overflows. It is an instrumentation tool, which requires a target binary to compile via *clang* with the *-fsanitize=address* option. After an instrumented JS engine crashes when running a given PoC, *AddressSanitizer* reports a memory-error class. Note that ClusterFuzz has used this class as an indicator for classifying new browser bugs [20].

Table 6 shows our evaluation results for *AddressSanitizer* on the collected PoCs and their crashes. The second column is a memory-error class reported by *AddressSanitizer*. For the first column, we

Table 6: Evaluation of *AddressSanitizer*

Our verdict	*AddressSanitizer*	Crash instances	
		Security	Not-security
Exploitable	stack-buffer-overflow	15 (6.47%)	**1 (0.47%)**
	heap-use-after-free	4 (1.72%)	**4 (1.87%)**
	stack-buffer-underflow	4 (1.72%)	**0 (0.00%)**
	invalid-free	1 (0.43%)	**2 (0.93%)**
	stack-use-after-return	1 (0.43%)	**0 (0.00%)**
	use-after-poison	0 (0.00%)	**1 (0.47%)**
Not-Exploitable	alloc-dealloc-mismatch	**16 (6.90%)**	10 (4.67%)
	memory-leaks	**1 (0.43%)**	3 (1.40%)
	stack-overflow	**0 (0.00%)**	5 (2.34%)
Ignored	invalid-memory-access	148 (63.79%)	143 (66.82%)
	not-segv	42 (18.10%)	45 (21.03%)

clustered the reported memory-error classes into three groups: *Exploitable*, *Not-exploitable*, and *Ignored*. For this clustering, we leveraged the criteria from ClusterFuzz [21]. It classifies the severity of a bug when *AddressSanitizer* emits one of the following labels: Bad-cast, Heap-buffer-overflow, Heap-double-free, Heap-use-after-free, Stack-buffer-overflow, Stack-use-after-return, or Use-after-poison. Otherwise, we referenced the Common Weakness Enumeration (CWE) list [39] to decide whether each memory-error class was exploitable. For instance, CWE-762 indicates alloc-dealloc-mismatch and its official CWE description states that exploiting this bug is rarely likely to cause unauthorized code execution [40]. We ignored 378 crash instances with the invalid-memory-access and not-segv labels when computing the precision, recall, accuracy, and false negatives of *AddressSanitizer* because we were unable to make further decisions whether or not they are security-related.

As Table 6 shows, *AddressSanitizer* achieved 0.76 precision. However, the accuracy is 0.63 and the recall is 0.60, thus producing 40% of false negatives. Also, *AddressSanitizer* was unable to detect 190 of the crashes (42%) triggering security bugs. It becomes evident that the tool is designed for detecting general memory errors, rather than for classifying security bugs in JS engines.

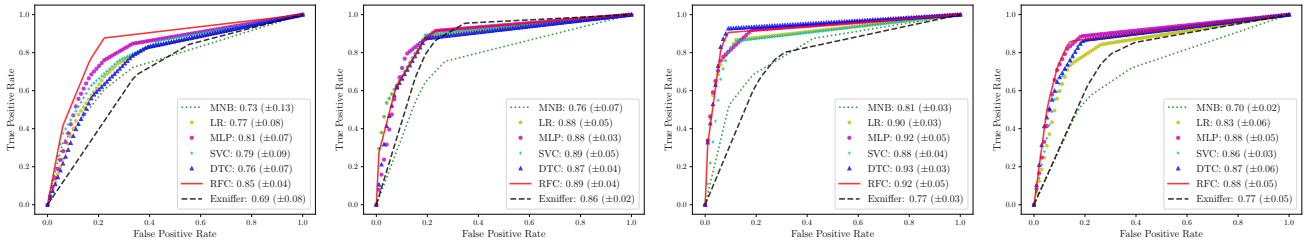
6.2 CRScope Performance (RQ2)

We evaluated CRScope on crash-dump files from Chakra, V8, and SpiderMonkey. Table 7 summarizes the averaged accuracy, precision, recall, F-1 score, and AUC for the model that appears in the "Selected model" row. We selected the model with the highest AUC among six classifiers. We also evaluated each of the classification models, which were trained and validated on crash-dumps only from each JS engine listed between the third and sixth head columns. "All" represents the evaluation result of the model with crash instances from all three JS engines.

Table 7 demonstrates that CRScope is effective in classifying security crashes of JS engines. For SpiderMonkey crashes, the DTC model of CRScope achieved 0.93 accuracy, 0.95 precision, 0.92 recall, 0.93 F1-score, and 0.93 AUC; this showed better performance than any of the other models. Figure 2 also shows the ROC curves of the six models for each JS engine. In general, CRScope is good at classifying security bugs when its model is trained on crash-dump files only from each JS engine. As Section 6.3 describes, the most

Table 7: Evaluation of CRScope with oversampled crash instances.

Target system	JS engines	Chakra	V8	SpiderMonkey	All
CRScope	**Selected model**	RFC	RFC	DTC	RFC
	Accuracy	0.85 ± 0.04	0.89 ± 0.04	0.93 ± 0.04	0.88 ± 0.05
	Precision	0.84 ± 0.06	0.89 ± 0.08	0.95 ± 0.03	0.89 ± 0.04
	Recall	0.87 ± 0.03	0.90 ± 0.02	0.92 ± 0.07	0.86 ± 0.07
	F1-Score	0.85 ± 0.04	0.89 ± 0.04	0.93 ± 0.04	0.87 ± 0.05
	AUC	0.85 ± 0.04	0.89 ± 0.04	0.93 ± 0.03	0.88 ± 0.05
Exniffer	**Selected model**	RFC	RFC	MLP	RFC
	Accuracy	0.69 ± 0.09	0.86 ± 0.02	0.77 ± 0.03	0.77 ± 0.05
	Precision	0.64 ± 0.08	0.81 ± 0.04	0.76 ± 0.02	0.74 ± 0.05
	Recall	0.83 ± 0.06	0.94 ± 0.06	0.78 ± 0.09	0.83 ± 0.06
	F1-Score	0.72 ± 0.07	0.87 ± 0.02	0.77 ± 0.04	0.78 ± 0.04
	AUC	0.69 ± 0.08	0.86 ± 0.02	0.77 ± 0.03	0.77 ± 0.05
CRScope + *Exniffer*	**Selected model**	RFC	DTC	DTC	DTC
	Accuracy	0.81 ± 0.08	0.91 ± 0.05	0.93 ± 0.02	0.88 ± 0.03
	Precision	0.80 ± 0.08	0.91 ± 0.05	0.95 ± 0.04	0.90 ± 0.05
	Recall	0.84 ± 0.09	0.87 ± 0.08	0.92 ± 0.06	0.87 ± 0.02
	F1-Score	0.82 ± 0.08	0.90 ± 0.06	0.93 ± 0.02	0.88 ± 0.03
	AUC	0.81 ± 0.08	0.91 ± 0.05	0.93 ± 0.02	0.88 ± 0.03



(a) Chakra **(b) V8** **(c) SpiderMonkey** **(d) All**

Figure 2: ROC curves for CRScope with oversampled crash instances

significant features included backtraces and function names, each of which differs among JS engines.

We also compared our experimental results for the models trained on the features addressed in *Exniffer* [60]. *Exniffer* is a machine learning based tool that predicts the exploitability of crashes using crash-dump files and the last branch record (LBR) register. This work used a total of 51 static and dynamic features from crash-dumps and the LBR register, respectively. Out of the 51 features, there were 44 static features, which consisted of whether the backtrace was corrupted; whether the instruction pointer, base pointer, and stack pointer referenced valid memories; the type of access violation; the number of operands; each operand type; and the type of signal. When implementing *Exniffer*, we used static (not dynamic) features to leverage only crash-dump files. We excluded LBR information because the extraction of this dynamic information from complex software, such as JS engines, is difficult for normal users who lack security expertise; this issue contradicts our goal. Furthermore, Tripathi *et al.* demonstrated that the LBR features were not significant when *Exniffer* classified exploitability; the LBR features did not appear in the top five significant features [60].

Table 7 shows that CRScope achieves a better performance than *Exniffer* in all cases. It demonstrates that the *Exniffer* features are not more effective than ours in determining security bugs. Interestingly, when combining the features from CRScope and *Exniffer*, the performance in all cases, except Chakra's one, is boosted. This result clearly demonstrates that CRScope features significantly contribute to identifying security bugs, and *Exniffer* features are complementary, partly contributing to CRScope performance improvement.

6.3 Feature Importance (RQ3)

Table 8 shows the list of the top five features, ranked by their significance in each selected model, which produced the best performance. Some listed features have a square bracket ([ ]), which represents the feature type and the vectorization method used to extract the feature. The left-hand element of a colon in the bracket is a feature type. We abbreviate *Crashing instruction*, *Crashing function*, and

Table 8: Most significant features in the CRScope models

CRScope		CRScope + *Exniffer*	
JS engines	**Top-5 features**	**JS engines**	**Top-5 features**
Chakra (RFC)	1. [b:c] js 2. [b:t] js profilinghelpers 3. [b:c] interpreterstackframe 4. [i:t] rbx 5. [b:t] javascriptproxy	**Chakra (RFC)**	1. [b:c] js 2. [b:t] js profilinghelpers 3. [i:t] rbx 4. [f:t] js sourcetextmodulerecord 5. [b:t] javascriptproxy
V8 (RFC)	1. [b:t] abort 2. signal 3. [i:t] ptr 4. [i:t] ud2 5. [f:t] v8	**V8 (DTC)**	1. signal 2. [f:c] accessors 3. [b:t] internal object 4. [b:c] compile 5. **[F13] Memory operand is Null**
Spider Monkey (DTC)	1. [b:c] runfile 2. [b:c] jit 3. [b:c] jsstring 4. [f:t] js 5. [i:t] 0x1cb	**Spider Monkey (DTC)**	1. [f:t] js 2. **[F34] Resume Flag** 3. [b:c] jit 4. [b:c] runfile 5. [b:c] gc
All (RFC)	1. [f:t] execute executescript 2. [f:t] js 3. [f:c] internal 4. crash_type 5. [b:t] js profilinghelpers	**All (DTC)**	1. [f:t] js 2. **[F20] Operand is register** 3. [b:c] internal 4. crash_type 5. [i:t] qword

Backtrace as "i," "f," and "b," respectively. The right-hand element indicates either *CountVectorizer* or *TfidfVectorizer*, which are abbreviated as "c," and "t," respectively. For instance, [i:t] ptr means that its corresponding feature value is ptr that is extracted from a crashing instruction by applying the *TfidfVectorizer* method. Other features without the square bracket represent a feature type extracted via *LabelEncoder*, such as *signal* and *crash_type*. The other features starting with "F" in a square bracket represent features from *Exniffer*. The number next to the "F" represents a feature index used in [60].

Chakra. The third row shows the top five significant features for the Chakra classifier models. When considering only CRScope features, most of the significant features, except for the fourth top feature, were namespace and class names extracted from backtraces. The first, second, third, and fifth top features represent the Js namespace, Js::ProfilingHelpers class, Js::InterpreterStackFrame class, and Js::JavascriptProxy class, respectively. We investigated the Chakra dataset and found that 88.1% of crash instances with the Js::ProfilingHelpers feature and 73.8% with the Js::InterpreterStackFrame feature were security bugs. On the other hand, all crash instances with the Js::JavascriptProxy feature were non-security bugs, thus enabling the classifier to assign a non-security bug for these crash instances. It is apparent that the classes appearing in a backtrace significantly contribute to the CRScope classification of security bugs.

The fourth feature is the rbx register extracted from the crashing instructions. We investigated the usage of rbx in our Chakra dataset. This register stored a data value (e.g., mov rbx, QWORD PTR [r12]) or referenced a memory address (e.g., mov rax, QWORD PTR [rbx+0x8]). Interestingly, when the rbx register references a non-null memory address, such crashes were classified as security-related. This result shows that the usage of the rbx register at a crashing instruction plays an important role in security bug classification.

V8. The top feature of the V8 model is abort, which is a 1-gram token of the v8::internal::OS::Abort or v8::base::OS::Abort functions extracted from backtraces. These functions are primarily utilized when the engine detects an error and then terminates by itself. In our V8 dataset, 68.6% of crash instances with the abort feature were non-security bugs. Interestingly, however, crash instances with abort feature had SIGILL (86.1%) or SIGABRT (13.9%) signals, and 73.7% of crash instances with SIGILL were non-security bugs, while 63.2% of crash instances with SIGABRT were security bugs. Therefore, this top feature contributes to CRScope identifying both security and non-security bugs.

The second top feature is *signal*. In our V8 dataset, a signal is one of SIGSEGV, SIGABRT, SIGILL, and SIGFPE. About 87.4% of crash instances with the SIGSEGV feature and 63.2% of crash instances with the SIGABRT feature were security bugs. On the other hand, all crash instances with the SIGFPE feature and 73.7% of crash instances with the SIGILL feature were non-security bugs. Hence, the former and latter signals affect the capability of CRScope to classify security bugs and non-security bugs, respectively.

The third and fourth top features were extracted from crashing instruction. Any instructions involving ptr read or write from/to main memory addresses. In our V8 dataset, 93% of crash instances with ptr had the SIGSEGV signal, which indicates a memory bug. Approximately 85.9% of crash instances with the ptr feature were security-related. The ud2 instruction means an undefined instruction and is closely related to the abort functions and SIGILL signal because all crash instances with the ud2 feature had the SIGILL signal and backtraces involving the abort functions. Therefore, it had the same percentage of non-security bugs with SIGILL crash instances (i.e., 73.7%). Among crashing instructions, ptr and ud2

```
function f() {
    ({a = () => {
        let arguments;
    }} = 1);

    arguments.x;
}
f();
```

Listing 1: Chakra PoC code that invokes the security bug of CVE-2017-8670

```
var proxy = new Proxy(function(){}, {});
class C extends proxy {
    constructor() {
        super(Object.setPrototypeOf(C, function(){}))
    }
}
Reflect.construct(C, [], proxy);
```

Listing 2: Chakra PoC code that invokes a non-security bug

contributed to CRScope being able to identify security bugs and non-security bugs, respectively.

SpiderMonkey. The runfile feature, the top feature for SpiderMonkey, indicates the partial sequence of a backtrace,

$$RunFile - Process - ProcessArgs - Shell - main,$$

which locates at the bottom of the backtrace. In our SpiderMonkey dataset, 83.1% of crash instances with that backtrace were non-security bugs. However, we concluded that this feature has no discriminative capability since the call sequence including runfile is a common method for executing JS tests, which does not have any security implication.

The second and third top features represent the js::jit and JSString classes, respectively. We analyzed the usage of those classes in our dataset. We found that the more the js::jit class appears in a backtrace, the more likely its crash instance is security-related. The rate of security bugs among the crash instances for which the number of the js::jit class appearing in the backtrace is greater than zero was 45.3%. This rate grows as the number of js::jit occurrences increases; the security bug rate when considering only backtraces for which the number of js::jit occurrences was greater than two increased to 71.4%. Further, the rate went up to 77.3% when the occurrence was greater than three, and 82.4% when the occurrence was greater than four. For the crashes with backtraces involving JSString, we observed two kinds of backtraces; JSString::isRope - JSString::isLinear - JSString::ensureLinear and JSString::dumpRepresentation. Interestingly, all crash instances with the former backtrace were non-security bugs, while the latter were security bugs.

The fifth top feature is the 0x1cb value extracted from crashing instruction. In our SpiderMonkey dataset, there were four crashing instruction instances involving 0x1cb, and they shared the same instruction semantic: mov DWORD PTR [eax], 0x1cb. This operation attempts to write 0x1cb to a memory location referenced by a register, resulting in a write violation when the referenced address is not accessible or allocated. We observed that all of these instances were security bugs.

Exniffer. When covering both CRScope and *Exniffer* features, our features showed up in the top five significant features in most cases. It indicates that CRScope features are more adequate in terms of labeling security bugs. Interestingly, when the AUC of a model covering both features increased, an *Exniffer* feature is included in the top five features. For the V8 model, the AUC is boosted from 0.89 to 0.91, and the top fifth feature includes an *Exniffer* feature, F13: Memory operand is Null. Also, the AUC of the SpiderMonkey model is slightly boosted from 0.93 ± 0.03 to 0.93 ± 0.02, and the top second feature includes F34: Resume Flag. Meanwhile, in Chakra, the AUC of the combination system is not boosted, and the top five features consist of only CRScope features. This result demonstrates that *Exniffer* provides complementary features for CRScope to improve.

Case Study. Listings 1 and 2 show Chakra PoC code snippets that invoke a security bug and a non-security bug, respectively. The following table shows two feature vectors representing two Chakra crash instances obtained by executing Listings 1 and 2. Each row represents one crash instance with two vector representations; the first one is from CRScope and the other one is encoded according to the method used by *Exniffer*.

CRScope	*Exniffer*
6, 0.101488, 4, 0., 0, ...	1, 2, 3, 4, 6, 8, 9, 5, 11, 16, 20, 18, 19, 23, 24, 27, 29, 34, 42
4, 0., 4, 0.243038, 0, ...	1, 2, 3, 4, 6, 8, 9, 5, 11, 16, 20, 18, 19, 23, 24, 27, 29, 34, 42

The CRScope feature dimension is 100, which is a spare vector with most element values are zero. On the other hand, the length of the *Exniffer* feature is 44, and the listed features are set to 1. When only considering *Exniffer* features, they are identical, although their corresponding PoC code completely differ. However, one of them represents the crash due to a security bug, and the other represents the crash due to a non-security bug. It is clear that *Exniffer* gives the same label to these two bugs. On the other hand, CRScope correctly labels the first bug as a security bug, and the other as a non-security bug because we encode each vector element with a numeric (quantitative) value, not a binary value, providing the models with more flexibility in classifying security bugs.

7 RELATED WORK

There are a limited number of previous studies on predicting exploitability [23, 60, 66] and prioritizing bugs by analyzing call stacks [6, 32, 64] or descriptions [2, 9, 31] in bug reports. No previous work addresses predicting security-related bugs solely from an observed JS engine crash. Predicting security-related bugs in a JS engine is not straightforward. In practice, classifying a reported bug requires domain expertise because of the complexity and enormous size of state-of-the-art JS engines. To our best knowledge, CRScope is the first tool to classify security-related bugs from crash-dumps by building machine learning models from the past verdicts of JS engine developers.

7.1 Crash Analyses

Gustavo *et al.* proposed *VDiscover* [23] to predict the exploitability of a given test using machine learning. They leveraged static and dynamic call sequences to the standard C library. Because their approach uses a single static call sequence for each binary, the static feature of *VDiscover* is not directly applicable for training each JS engine that emits different crash-dump files for various

JS inputs. They also required instrumenting a target binary and its dynamically linked libraries to extract dynamic call sequences, resulting in significant overheads for large software such as JS engines.

Exniffer [60] also classifies the exploitability of a given core-dump by using machine learning models, similar to the approach used by CRScope. *Exniffer* extracts a set of features from core-dump files and run-time information by leveraging hardware-assisted monitoring such as Last Branch Record (LBR) register [24]. In contrast to monitoring execution traces, LBR does not impede any real-time performance. However, it has a limited window size for tracing executed instructions, and requires an Intel processor with LBR supports. Without LBR support, an LBR simulator is required, which brings inevitable performance overhead. The total number of features used in *Exniffer* was 51, which is seven times larger than the number of features used in CRScope. This difference comes from the method used to vectorize each feature, not from a limited feature scope of CRScope. *Exniffer* encodes every feature value as binary number, thereby increasing feature vector dimensions.

ExploiMeter [66] uses fuzzing and machine learning techniques to evaluate software exploitability. It also uses static features from ELF executables extracted using hexdump, objdump, ldd, and readelf utilities. Although *ExploiMeter* uses dynamic fuzzing tests to update the prior beliefs on exploitability, the dataset is labeled using *!exploitable*, which is shown to be less accurate than *Exniffer*.

RETracer [5] is another study that identifies functions to blame for an observed crash. It analyzes program semantics extracted from memory dumps. It performs backward and forward taint analyses, then identifies blamed functions from backward data flow graphs. Although this work was deployed on Windows Error Reporting (WER), it focused on the Windows platform running on x86 and x86-64 architectures. Also, they did not consider whether a given crash is due to a security bug or not.

The representative difference of CRScope from the previous studies is that we classify security bugs, not exploitable bugs, and target JS engines, which are larger and more sophisticated than the common software projects that the previous studies were evaluated on. Furthermore, CRScope uses only crash dumps which are trivial to obtain.

7.2 Bug Report Analyses

Classifying bug reports is important for the BTSs of software projects like browsers to prioritize bugs, detect duplicates, and assign developers. Various research efforts have been devoted to triaging bug reports in BTSs.

Stack Traces. This line of work [6, 32, 64] presented a method for prioritizing crashes using stack traces, although no consideration was given to whether the crash was exploitable or security-related. Kim *et al.* [32] predicted whether a crash will be frequent (top crash) or not (bottom crash). They extracted crash stack traces and functions to train their model. Dang *et al.* [6] improved existing Microsoft Windows Error Reporting (WER) by proposing a novel bucketing method based on the Position Dependent Model (PDM), which is a similarity measure for call stacks. *CrashLocator* [64] recovers approximate crash traces via stack expansions and computes the suspicious scores for each function in the recovered traces. A ranked list sorting the functions by their suspicious scores helps to locate a faulty function. These approaches using stack traces are simple and bring almost no overhead; however, they are not effective for newly added functions.

Descriptions. In [2, 9], Gegick *et al.* and Behl *et al.* proposed approaches that apply text mining on natural-language descriptions of bug reports to train a statistical model for classifying a bug report as either a security bug report or a non-security bug report. Kanwal and Maqbool developed a recommender which automatically prioritizes new bug reports using machine learning [31]. They used the categorical and text attributes of a bug report as training features. The former included, for example, component, severity, platform, operating system, bug lifetime, developer, and the latter included the summary and description. Some research introduced machine learning approaches to detect duplicate bug reports [28, 52, 59] and other presented machine learning approaches to recommend bugs that the developer should work on [1, 29, 50]. Although the previous research used bug reports from large-scale software projects such as Eclipse, Cisco, and Mozilla, their main limitation is that they require well-written bug reports created by the bug reporters.

8 CONCLUSION

We designed and implemented CRScope to classify whether a given crash-dump is security-related. Specifically, it checks whether the PoC code snippets causing JS engine crashes trigger its inherent security bugs. We also demonstrated that prior approaches, including *Exploitable* and *AddressSanitizer*, are unfit for classifying security bugs in JS engines.

CRScope leverages a machine learning classifier trained on past verdicts by domain experts. Rather than using arbitrarily selected general features, we feed the model with engine-specific local features, which reflect the historical context in which each JS engine crashed. We then let CRScope select the best features for rendering correct verdicts.

When evaluated on 339 bugs and their 766 crashes, CRScope achieved 0.85, 0.89, and 0.93 AUCs for Chakra, V8, and SpiderMonkey, respectively; it outperforms all tools including *Exniffer* from the previous studies in all cases. The experimental results demonstrate its practical utility. We invite further research by releasing the ground truth dataset and the source code of CRScope.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their concrete feedback. This work was supported by National Research Foundation of Korea (NRF) Grant No.: 2017R1C1B5073934 and by the Naver corporation.

REFERENCES

[1] John Anvik and Gail C Murphy. 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 3 (2011), 10.
[2] Diksha Behl, Sahil Handa, and Anuja Arora. 2014. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on.* IEEE, 294–299.
[3] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson Correlation Coefficient. In *Noise Reduction in Speech Processing.* Springer, 1–4.
[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort,

Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[5] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. 2016. RETracer: triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 820–831.

[6] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 1084–1093.

[7] CVE Details. 2019. Vulnerability Statistics Of Chrome. https://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224.

[8] Jonathan Foote. 2018. GDB 'exploitable' plugin. https://github.com/jfoote/exploitable.

[9] Michael Gegick, Pete Rotella, and Tao Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *Mining software repositories (MSR), 2010 7th IEEE working conference on*. IEEE, 11–20.

[10] Google. 2016. Chromium Issue 388665. https://bugs.chromium.org/p/chromium/issues/detail?id=388665.

[11] Google. 2016. Chromium Issue 595834. https://bugs.chromium.org/p/chromium/issues/detail?id=595834.

[12] Google. 2018. Chromium Issue 386988. https://bugs.chromium.org/p/chromium/issues/detail?id=386988.

[13] Google. 2018. Chromium Issue 610600. https://bugs.chromium.org/p/chromium/issues/detail?id=610600.

[14] Google. 2018. Chromium Issue 718858. https://bugs.chromium.org/p/chromium/issues/detail?id=718858.

[15] Google. 2018. Chromium Issue 729991. https://bugs.chromium.org/p/chromium/issues/detail?id=729991.

[16] Google. 2018. Chromium Issue 733549. https://bugs.chromium.org/p/chromium/issues/detail?id=733549.

[17] Google. 2018. Reporting Security Bugs. https://dev.chromium.org/Home/chromium-security/reporting-security-bugs.

[18] Google. 2019. Chrome Reward Program Rules. https://www.google.com/about/appsecurity/chrome-rewards/.

[19] Google. 2019. Chromium Issues. https://bugs.chromium.org/p/chromium/.

[20] Google. 2019. ClusterFuzz. https://github.com/google/clusterfuzz.

[21] Google. 2019. ClusterFuzz Crash type. https://google.github.io/clusterfuzz/reference/glossary.

[22] Google. 2019. Google V8. https://chromium.googlesource.com/v8/v8/.

[23] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 85–96.

[24] Part Guide. 2016. Intel 64 and IA-32 architectures software developer's manual. *Volume 3 (3A, 3B, 3C & 3D): System Programming Guide* (2016).

[25] Mark Andrew. Hall. 1999. Correlation-Based Feature Selection for Machine Learning. (1999).

[26] Choongwoo Han. 2019. Case Study of JavaScript Engine Vulnerabilities. https://github.com/tunz/js-vuln-db.

[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. 445–458.

[28] Nicholas Jalbert and Westley Weimer. 2008. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 52–61.

[29] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 111–120.

[30] Jeffrey D. Ullman Jure Leskovec, Anand Rajaraman. 2014. Mining of Massive Datasets. http://infolab.stanford.edu/~ullman/mmds/book.pdf.

[31] Jaweria Kanwal and Onaiza Maqbool. 2012. Bug prioritization to facilitate bug report triage. *Journal of Computer Science and Technology* 27, 2 (2012), 397–412.

[32] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. 2011. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering* 37, 3 (2011), 430–447.

[33] Kaspersky Lab. 2018. Attacks leveraging exploits for Microsoft Office grew fourfold in early 2018. https://www.kaspersky.com/about/press-releases/2018_microsoft-office-exploits.

[34] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. 2017. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research* 18, 17 (2017), 1–5. http://jmlr.org/papers/v18/16-365.html

[35] Microsoft. 2014. Definition of a Security Vulnerability. https://docs.microsoft.com/en-us/previous-versions/tn-archive/cc751383(v=technet.10).

[36] Microsoft. 2019. EdgeHTML issue tracker. https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/.

[37] Microsoft. 2019. Microsoft ChakraCore. https://github.com/Microsoft/ChakraCore.

[38] Microsoft. 2019. Microsoft Edge on Windows Insider Preview Bounty Program. https://www.microsoft.com/en-us/msrc/bounty-edge.

[39] MITRE. 2018. Common Weakness Enumeration. https://cwe.mitre.org/.

[40] MITRE. 2019. CWE-762. https://cwe.mitre.org/data/definitions/762.html.

[41] Mozilla. [n. d.]. Mozilla Bug Bounty Program. https://www.mozilla.org/en-US/security/bug-bounty/.

[42] Mozilla. [n. d.]. Mozilla Client Bug Bounty Program. https://www.mozilla.org/en-US/security/client-bug-bounty/.

[43] Mozilla. 2017. Mozilla Bug 1344415. https://bugzilla.mozilla.org/show_bug.cgi?id=1344415.

[44] Mozilla. 2018. Mozilla Bug 1493900. https://bugzilla.mozilla.org/show_bug.cgi?id=1493900.

[45] Mozilla. 2018. Mozilla Bug 1493903. https://bugzilla.mozilla.org/show_bug.cgi?id=1493903.

[46] Mozilla. 2019. Bugzilla. https://bugzilla.mozilla.org/.

[47] Mozilla. 2019. Exploitable crashes. https://developer.mozilla.org/en-US/docs/Mozilla/Security/Exploitable_crashes.

[48] Mozilla. 2019. Mozilla SpiderMonkey. https://github.com/mozilla/gecko-dev.

[49] Mozilla. 2019. Security Advisories for Firefox. https://www.mozilla.org/en-US/security/known-vulnerabilities/firefox/.

[50] G Murphy and D Cubranic. 2004. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer.

[51] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 529–540.

[52] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 70–79.

[53] Jesse Ruderman. 2007. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[54] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 118–121.

[55] Offensive Security. 2019. Exploit Database. https://www.exploit-db.com/.

[56] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*. 309–318.

[57] Internet World Stats. 2019. INTERNET USAGE STATISTICS. https://www.internetworldstats.com/stats.htm.

[58] ChakraCore team. 2019. ChakraCore Roadmap. https://github.com/Microsoft/ChakraCore/wiki/Roadmap.

[59] Yuan Tian, Chengnian Sun, and David Lo. 2012. Improved duplicate bug report identification. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 385–390.

[60] Shubham Tripathi, Gustavo Grieco, and Sanjay Rawat. 2017. Exniffer: Learning to Prioritize Crashes by Assessing the Exploitability from Memory Dump. In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 239–248.

[61] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.

[62] W3Techs. 2019. Usage of JavaScript for websites. https://w3techs.com/technologies/details/cp-javascript/all/all.

[63] Webkit. 2019. Webkit JavaScriptCore. https://git.webkit.org/.

[64] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 204–214.

[65] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 529–540.

[66] Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. 2017. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. IEEE, 164–175.

[67] ZERODIUM. 2019. ZERODIUM Exploit Acquisition Program. https://zerodium.com/program.html.

[68] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware Performance Counters Can Detect Malware: Myth or Fact?. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 457–468.