Crash Report Prioritization for Large-Scale Scheduled Launches

Nimmi Rashinika Weeraddana
University of Waterloo, Canada
nrweeraddana@uwaterloo.ca

Sarra Habchi
Ubisoft La Forge, Canada
sarra.habchi@ubisoft.com

Shane McIntosh
University of Waterloo, Canada
shane.mcintosh@uwaterloo.ca

***Abstract*—Software crashes are high-impact bugs that cause applications to terminate unexpectedly. Crash prioritization focuses the attention of maintenance teams on incoming types of crashes that are likely to have a large impact. Prior approaches have been applied to software that is continuously released, whereas, in the context of video game development, large releases are often rolled out following a schedule. A game studio will work for months or years on a new title before releasing it on a scheduled date. In that context, crash data from live players is not available until after release, which is often too late to react.**

In this study, we analyze post-release game crashes to identify temporal patterns that can inform strategies for prioritization at Ubisoft—a multinational video game publisher. Our analysis shows that most types of post-release crashes impact few players; however, a subset goes "viral," quickly impacting many players. Those viral crashes may escalate immediately (i.e., outbreaks) or lie dormant before propagating to many players (i.e., time bombs). We use data from a previously released title to detect such viral crashes in a new title by leveraging stack trace similarity and Machine Learning (ML). We found that prioritizing crash types based on stack trace similarity outperforms prioritization using ML models, successfully identifying over half of the viral crash types in the new title. Moreover, the crash types detected by our similarity-based prioritization account for a significantly larger number of crash occurrences than the viral crash types that are misclassified as non-viral. The findings of our study inform game development teams about proactive monitoring of mild crash types that can potentially impact many players, as well as approaches for early detection of potential viral crash types.

***Index Terms*—crash prioritization, viral crashes**

I. Introduction

Software crashes are disruptive faults [42] that abruptly terminate software applications.[1] Crashes cause user frustration and system outages [35]. A crash may occur due to a variety of root causes. For example, a specific combination of untested user actions may cause software crashes [1]. Software may also crash due to bugs and/or constraints in external resources [27]. For example, millions of systems running Microsoft Windows crashed due to a defective patch in an external dependency.[2]

In large-scale software systems, where thousands of crashes are reported daily, critical crashes are often addressed [24], but crashes that are deemed less important remain unresolved due to constraints on developer time or an inability to reproduce the crash in-house. In response to this, previous studies have proposed approaches to prioritize crashes. For example, Kim et al. [14] used a dataset of crashes reported by Mozilla's Crash Reporter[3] to predict the most frequent crash types by training Machine Learning (ML) models. This approach enables faster resolution of the potentially prevalent crash types in Mozilla Firefox with 68–80% accuracy. One limitation of this method is that it relies on historical data from previous releases to predict prevalent crash types in new releases.

Unlike software such as Mozilla Firefox, which follows a frequent release schedule (e.g., every six weeks[4]), video games are often released in large-scale, singular launches after months or years of development. These launches are followed by patch releases to fix bugs or introduce new content, such as seasonal updates. In a large-scale video game launch, it is not always possible to learn from previous releases to anticipate prevalent crash types. The popularity of a game also varies over time, with a majority of players being active during the initial launch period. Thus, the initial launch period is both the most active and the riskiest, as live player data is only beginning to accrue.

In this paper, we analyze temporal patterns of crash occurrences to inform approaches to prioritize crash types that are likely to propagate virally and impact many players. Moreover, we propose a similarity-based prioritization and an ML-based prioritization to leverage crash data from previously published titles to anticipate prevalent crash types in a new title. We structure the paper along reporting on these two analyses:

Patterns of crash occurrences (Sec. IV). Understanding how crash occurrences accumulate for each crash type is crucial for identifying the crash types that require immediate attention, particularly those with the potential to impact a large segment of the player base. To detect such temporal patterns, we apply clustering approaches to the time series of crash occurrences of crash types in Project G1—a high-budget (a.k.a., triple-A) title produced by Ubisoft.

Results. Three temporal patterns dominate in the Title G1 dataset. First, we observe *mild* crash types that impact a few players; indeed, the majority of crashes follow this mild pattern. Second, we observe *outbreaks*, which rapidly proliferate

[1]https://support.mozilla.org/en-US/kb/troubleshoot-firefox-crashes-closing-or-quitting

[2]https://www.bnnbloomberg.ca/business/company-news/2024/07/20/crowdstrike-crash-affected-85-million-microsoft-windows-devices/

[3]https://crash-stats.mozilla.org/

[4]https://blog.mozilla.org/futurereleases/2011/07/19/every-six-weeks/

shortly after the launch of the title, impacting many players. Third, we observe *time bombs*, which remain dormant for a period after a release before propagating to impact many players. Based on these patterns, we collectively refer to outbreaks and time bombs as "viral" crash types, and strive to help in prioritizing them before a title is released.

Crash type prioritization (Sec. V). Fixes for crash types that are likely to be viral should be prioritized because they have a disruptive impact on the player experience that also propagates to many players. To detect such crash types using stack trace similarity, we obtain Large Language Model (LLM)-based embeddings of stack traces and compute the similarity between crash types from a previous title and a new title. Any crash type from the new title that has a similar crash type that was viral in a previous title is flagged using this similarity-based prioritization. We also train ML models on crash data from previous titles to detect crash types in a new title that are likely to grow virally in the initial release period.

Results. Our similarity-based prioritization outperforms the ML-based prioritization in detecting viral crash types in the new title. For instance, the Area Under the Receiver Operating Characteristics (AUROC) for our similarity-based prioritization is 0.76, outperforming our ML-based prioritization by 17 percentage points. Moreover, our similarity-based prioritization detects 53% of the pre-release crash types that grow virally shortly after the launch of the new title. Combining similarity-based and ML-based prioritization approaches does not show a substantial improvement. For instance, we introduce a combined approach where a crash type is classified as viral only if both the similarity-based and ML-based prioritization approaches agree. Otherwise, it is classified as non-viral; this combined approach achieves an AUROC of 0.57, which is 19 percentage points lower than the performance of the similarity-based prioritization alone. An inspection of the false positives (i.e., the crash types that are predicted to be viral but are non-viral in reality) that are produced by our similarity-based prioritization reveals that 11% of them accrue at least half of the threshold of total crash occurrences for viral crashes. An inspection of the false negatives (i.e., the crash types predicted to be non-viral but are viral in reality) suggests that the crash types classified as true positives accrue a significantly larger number of occurrences than those classified as false negatives (Mann-Whitney U test, unpaired, one-tailed, α = 0.05).

II. Crash Reporting and Challenges

Crash reporting systems help organizations to track crashes that occur in the field [15], [21], [45]. Several organizations have developed their own crash reporting tools, e.g., Mozilla's Crash Reporter[3] and Microsoft's Windows Error Reporting (WER) tool.[5] A crash report contains a stack detailing where program execution halted, as well as metadata about the timing of the crash and the environment in which it occurred. Crash

[5]https://learn.microsoft.com/en-us/troubleshoot/windows-client/system-management-components/windows-error-reporting-diagnostics-enablement-guidance

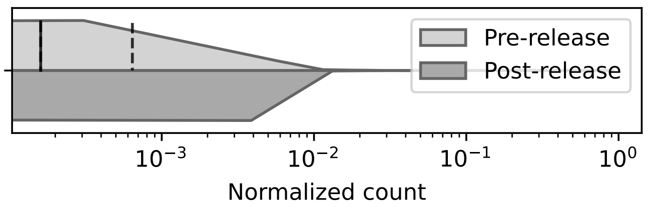


Fig. 1: Violin plot showing the distribution of the normalized number of crash occurrences per crash type reported in pre-release and post-release. X-axis is in a log scale. The solid lines represent the medians, while the dotted lines represent the first and third quartiles.

reporting systems use such data to group crash reports into distinct crash types [3], [28], [29], [32]. For example, Mozilla's Crash Reporter groups crash reports based on the method signatures of the top frames in their stack traces.[3]

A. The Crash Reporting System at Ubisoft

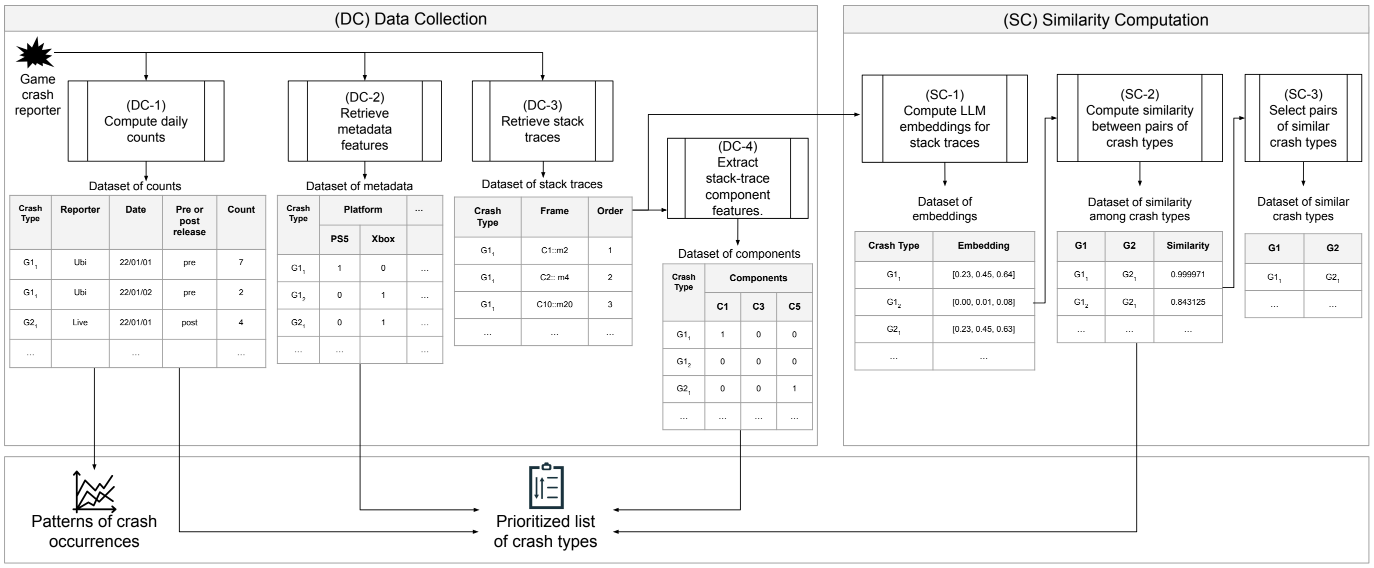
Ubisoft operates a proprietary crash reporting system designed to capture and analyze exceptions that cause the game to terminate unexpectedly. These exceptions arise from various issues, such as deadlocks and out-of-memory errors. The system records critical information about each crash event, including the type of the exception raised, the associated stack trace, and contextual metadata. This metadata encompasses details about the platform on which the crash occurred, the date and time of the event, the reporter type (e.g., live players, internal staff, or bots), and other relevant attributes.

Based on the stack trace, this system groups crashes into crash types. A new crash type is created each time a crash report cannot be categorized under any existing crash type. Each crash type can encompass multiple crash reports—some occurring during the pre-release phase (before a title's official launch) and others during the post-release phase (after a title has been officially launched). Crash types may span both phases.

B. The Crash Prioritization Problem in Large-Scale Scheduled Launches

While crash reporting systems group crash reports, to the best of our knowledge, built-in support for prioritization of crash types is not yet available. Approaches have been proposed in the literature, but most organizations build their own solution for crash prioritization. For example, Kim et al. [14] proposed a method to predict crashes based on previous versions of the same software. This approach is not directly applicable in the context of game development, where often only one large-scale scheduled release is produced for each title.

While prioritizing crash types that are prevalent among developers and testers during the pre-release phase is beneficial, the distribution of crash occurrences by crash type in the pre-release phase differs from that of the post-release phase [14]. To assess the difference in the number of crash occurrences between the pre- and post-release phases, we


(DC) Data Collection
Game crash reporter
(DC-1) Compute daily counts
(DC-2) Retrieve metadata features
(DC-3) Retrieve stack traces
(DC-4) Extract stack-trace component features.
Dataset of counts
Dataset of metadata
Dataset of stack traces
Dataset of components
(SC) Similarity Computation
(SC-1) Compute LLM embeddings for stack traces
(SC-2) Compute similarity between pairs of crash types
(SC-3) Select pairs of similar crash types
Dataset of embeddings
Dataset of similarity among crash types
Dataset of similar crash types
Patterns of crash occurrences
Prioritized list of crash types

Dataset of counts

Crash Type	Reporter	Date	Pre or post release	Count
$G1_1$	Ubi	22/01/01	pre	7
$G1_1$	Ubi	22/01/02	pre	2
$G2_1$	Live	22/01/01	post	4
…		…		…

Dataset of metadata

Crash Type	Platform		…
	PS5	Xbox	
$G1_1$	1	0	…
$G1_2$	0	1	…
$G2_1$	0	1	…
…	…		…

Dataset of stack traces

Crash Type	Frame	Order
$G1_1$	C1::m2	1
$G1_1$	C2:: m4	2
$G1_1$	C10::m20	3
…	…	…

Dataset of components

Crash Type	Components		
	C1	C3	C5
$G1_1$	1	0	0
$G1_2$	0	0	0
$G2_1$	0	0	1
…	…	…	…

Dataset of embeddings

Crash Type	Embedding
$G1_1$	[0.23, 0.45, 0.64]
$G1_2$	[0.00, 0.01, 0.08]
$G2_1$	[0.23, 0.45, 0.63]
…	…

Dataset of similarity among crash types

G1	G2	Similarity
$G1_1$	$G2_1$	0.999971
$G1_2$	$G2_1$	0.843125
…	…	…

Dataset of similar crash types

G1	G2
$G1_1$	$G2_1$
…	…

Fig. 2: An overview of our study design.

present an analysis based on data from Title G1—a triple-A game published by Ubisoft.

We first obtain the total number of crash occurrences for each crash type from the crash reporting system that were reported by developers and testers before the release. We apply min-max normalization to the number of pre-release crash occurrences across different crash types. Then, we compute the min-max normalized counts of crash occurrences for those same crash types reported by players after the release. Then, we perform a Wilcoxon rank-sum test (paired, two-tailed, α = 0.05) to test the following hypothesis.

Null hypothesis for pre- and post-release crash counts

The normalized number of crash occurrences reported in the pre-release phase is sampled from the same population as the normalized number of crash occurrences reported by live players.

Our statistical test yields a p-value of less than 0.05 with a large effect size (Cliff's $|\delta|$ = 0.93), allowing us to reject the null hypothesis. This indicates that the normalized distribution of crash occurrences in the pre-release phase, reported by developers and testers, is significantly different from the post-release crash occurrences reported by live players. This result is further supported by a violin plot shown in Fig. 1. Both the first quartile and the median of the normalized number of crash occurrences in the pre-release period is 1.6 $\times 10^{-4}$, while the third quartile is 6.4 $\times 10^{-4}$. On the other hand, the normalized number of post-release crash occurrences have the first quartile, median, and third quartile all substantially less than 10^{-4}. Due to this significant difference between the distributions of pre- and post-release crash occurrences of crash types, it is challenging to identify potential viral crash types before a title is released.

III. Study Design

In this section, we describe our procedures for Data Collection (DC) and Similarity Computation (SC) to analyze patterns of crash occurrences and prioritize crash types. Fig. 2 provides an overview of the process, which is detailed below.

(DC) Data Collection

To perform our study, we extract data from triple-A titles G1 and G2 published by Ubisoft. These titles were developed using the same game engine, and thus share commonalities. As G1 was released three years prior to G2, we strive to leverage insights from G1 to predict high-impact crashes in G2. The organization's crash reporting system collects both pre- and post-release crashes for both G1 and G2. Fig. 2 (DC) shows an overview of our data collection procedure. Below, we describe each step.

(DC-1) Compute daily counts. For each crash type reported in the pre-release phase, we calculate the number of daily crash occurrences reported by the internal staff and bots. For each crash type reported in the post-release phase, we calculate the number of daily crash occurrences reported by live players.

(DC-2) Retrieve metadata features. From the crash reporter, we extract detailed metadata for each crash type, such as the platform on which the crash occurred (e.g., Microsoft Xbox) and the operation that caused the crash (e.g., dereferencing a null pointer). We use the data from the first occurrence for each crash type because we strive to predict whether a crash type is likely to go viral after its first occurrence.

All of our metadata features are categorical in nature. Thus, we perform a one-hot vector encoding for each metadata feature.[6] Suppose the crash occurrences in our dataset come from either the Sony PlayStation 5 (PS5) or the Microsoft Xbox (Xbox) platform. In this case, for a crash type $G1_1$ in

[6]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

G1, in which the first occurrence in the post-release period occurred on PS5, our one-hot vector representation of the platform would be `[1,0]`. On the other hand, for a crash type $G2_1$, which first occurred on Xbox, our one-hot vector representation of the platform would be `[0,1]`.

(DC-3) Retrieve stack traces. The Ubisoft crash reporting system stores the stack trace for each crash occurrence, with each stack trace comprising multiple frames. Since the stack trace is identical for all occurrences of the same crash type, we use the stack trace from the first occurrence of each crash type as its representative. Then, we filter out the frames related to the crash reporting system itself, as these are not relevant to the underlying causes. The operations team at Ubisoft has found that the top three frames (i.e., *order* $\in$ $[1,2,3]$) in a stack trace are most often responsible for crashes. Hence, we extract the top three frames from each stack trace for further investigation. Focusing on the top three frames was informed by an analysis of common crash points in the projects under study. Other organizations may need to adjust the frame filter to align with the characteristics of their systems.

(DC-4) Extract stack trace component features. For each procedure present in the frame, we extract the associated component. For instance, if the procedure is represented as `C1::m2`, we identify `C1` as the component and `m2` as the method name. For procedures outside of the scope of a class, and without an explicitly defined component, we set its component to `global`. This distinction enables a component-level analysis of the crash origin. Below, we describe how component features are extracted for the two titles G1 and G2.

For G1, we select the top n components that appear most frequently across crash types and perform a bag-of-words encoding [13]. Suppose $n = 3$ and that `C1`, `C3`, and `C5` are the three components that appear most frequently in the crash types of G1. For each crash type in G1, we check if it implicates `C1`, `C3`, and/or `C5`. Suppose that the stack trace for a crash type $G1_1$ comprises the following three frames: `C1::m2`, `C2::m4`, and `C10::m20`. The bag-of-words encoding of $G1_1$ would be `[1,0,0]`.

For G2, we use the same set of top components that we extract from G1 to obtain bag-of-words encodings of components. For example, suppose $G2_1$ is a crash type with stack trace frames `C1::m3`, `C7::m7`, and `C5::m21`. The bag-of-words encoding for $G2_1$ components would be `[1,0,1]`.

For our study, we use $n = 400$ to cover a wide range of components while keeping the number of degrees of freedom spent under the recommended budget approximation of Lakshmanan et al. [16]. According to Lakshmanan et al., given f input features and C classes to classify, f should contain fewer than $\frac{|\text{crash types}|}{10\times\text{C}}$ degrees of freedom to mitigate the risk of *overfitting*, i.e., producing a model too specific to the training data to provide meaningful recommendations in production. For heavily imbalanced classes, as in our case, this budget should be further constrained, and is approximately 600.

(SC) Similarity Computation

To find similar crash types across the two titles, we compute stack trace similarity. Fig. 2 (SC) shows an overview of this procedure, which we detail below.

(SC-1) Compute LLM embeddings for stack traces. We use CodeLlama-7b [31]—an LLM specifically designed for code tasks—to extract embeddings for each crash type in G1 and G2. Each embedding is a 4096-dimensional vector, representing the stack trace of a crash type.

(SC-2) Compute similarity between pairs of crash types. For each pair of crash types in our dataset across the two titles, we compute the cosine similarity of CodeLlama embeddings of the stack traces. The cosine similarity between two crash types with embeddings A and B is $\frac{A\cdot B}{\|A\|\|B\|}$. We conjecture that this embeddings-based similarity measure provides a more effective measure of similarity between crash types than more naïve approaches that capture similarities in flat sequences, such as Hamming [11] and Levenshtein distance measures [37], which are likely to overlook semantic and structural similarities that exist among stack traces.

Suppose there are crash types $G1_1$, $G1_2$, and $G2_1$ with embeddings `[0.23, 0.45, 0.64]`, `[0.00, 0.01, 0.08]`, and `[0.23, 0.45, 0.63]`, respectively. We compute the pairwise similarity between crash types in G1 and G2, i.e., similarity for ($G1_1$, $G2_1$) is 0.99, and the similarity for ($G1_2$, $G2_1$) is 0.84. In this example, we use 3-dimensional embeddings, whereas in the actual dataset, we use 4096-dimensional embeddings.

(SC-3) Select pairs of similar crash types. To identify pairs of similar crash types, we must first select an appropriate similarity threshold t. To do so, we inspect 400 pairs of crash types with similarity scores above 0.95. In particular, for each pair of crash types, we examine whether the stack traces contain procedures from similar components and whether the method names are similar (though not necessarily identical). We find that pairs with similarity scores exceeding 0.99 exhibit the most similarity compared to those with lower thresholds. Therefore, we set $t = 0.99$ for further analysis. Following the example in Sec. III (SC-2), the pair of crash types ($G2_1$, $G1_1$) has a similarity score $\geq t$, and thus, we identify it as a pair of similar crash types.

IV. Patterns of Crash Occurrences

In this section, we explore the temporal patterns of crash occurrences by applying time-series clustering on crash data.

A. Approach

We apply TimeSeriesKMeans algorithm[7] to the time series of the cumulative count of post-release crash occurrences (as reported by live players) of crash types in G1. This algorithm requires a K setting to indicate the number of clusters. To tune the K setting, we compute the silhouette score [30], where scores above zero (ideally close to one) indicate well-defined clusters. The setting of K that maximizes

[7]https://tslearn.readthedocs.io/en/stable/gen_modules/clustering/tslearn.clustering.TimeSeriesKMeans.html

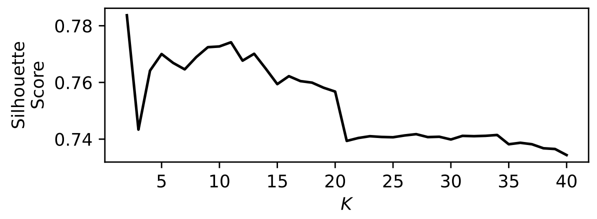


Fig. 3: Silhouette scores against a range of K settings.

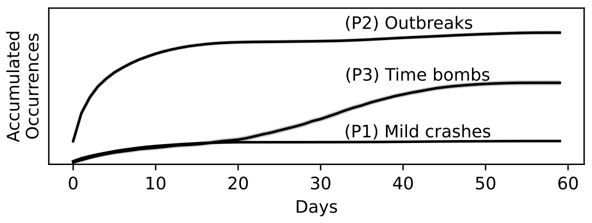


Fig. 4: Patterns of crash types. The Y-axis is logarithmically scaled.

the silhouette score is considered optimal. We explore settings of K that range between two and 40, and observe silhouette scores between 0.73–0.78. Fig. 3 shows the variation of the silhouette score for different K settings. From the figure, we observe that $K = 2$ and $K = 11$ settings have the highest silhouette scores [4]. For this study, we select $K = 11$ for a finer-grained analysis of crash patterns than $K = 2$. We obtain clusters for setting $K = 11 \pm 2$, and observe similar patterns.

B. Results

Fig. 4 shows the three most prevalent patterns among the generated time-series clusters. We include the rest of the patterns in our online appendix.[8] The Y-axis in the figure is scaled logarithmically. For each pattern, the black line represents the centroid or the mean shape of the group, while the shaded area indicates the 95% confidence interval.

Observation 1: Most types of post-release crashes impact only a few players (P1), whereas crash types that affect many players either escalate immediately after the release (P2) or remain dormant before propagating to many players (P3). Below, we discuss these three patterns in detail.

(P1) Mild crash types. These crash types accumulate a few occurrences over time. The time-series pattern P1 shown in Fig. 4 illustrates the mean shape of crash types that do not exhibit substantial growth in frequency after the release. These crash types are considered *mild* because they are encountered by only a small proportion of live players.

(P2) Outbreaks. These crash types rapidly increase in frequency shortly after release. The time series pattern P2 in Fig. 4 tends to grow quickly shortly after release. We refer to such crash types as *outbreaks*, and such rapid growth in frequency makes these crash types noticeable within days after the release, drawing the urgent attention of development teams to prevent further propagation.

(P3) Time bombs. These crash types initially grow slowly after the release but rapidly escalate later. The pattern P3 in Fig. 4 depicts the behavior of such crash types over time. During the first days after a release, these crash types typically accrue few occurrences, deceptively mimicking the behavior of P1. Yet, within a few days after the release, the number of occurrences can escalate quickly. We refer to such crash types as *time bombs* because they remain dormant during the initial days after the release before they propagate to a broad set of players.

[8]https://github.com/Ubisoft-LaForge/ubisoft-laforge-CrashReportPriorizationForLargeScaleScheduledLaunches

The emergence of these time-series patterns indicates that the importance of a crash type substantially varies over time. This leads us to conclude that viral crash types should be determined not only by the total number of occurrences but also by the timing and pattern of sudden surges in frequency.

V. Crash Type Prioritization

In this section, we explore approaches to prioritize crash types that are likely to be viral in the post-release phase. Specifically, we leverage data from Title G1 to prioritize crash types in Title G2 with the goal of ranking as many viral crash types in G2 as possible without overwhelming developers. Therefore, our objective is to achieve a balance between recall and precision.

A. Approach

Fig. 5 provides an overview of our approach to prioritizing viral crash types. Specifically, we present our approaches to Labeling Ground Truth (LG), Prioritization Based on Similarity (PS), and Prioritization Based on ML (PM). We then describe our Evaluation (E) of these approaches.

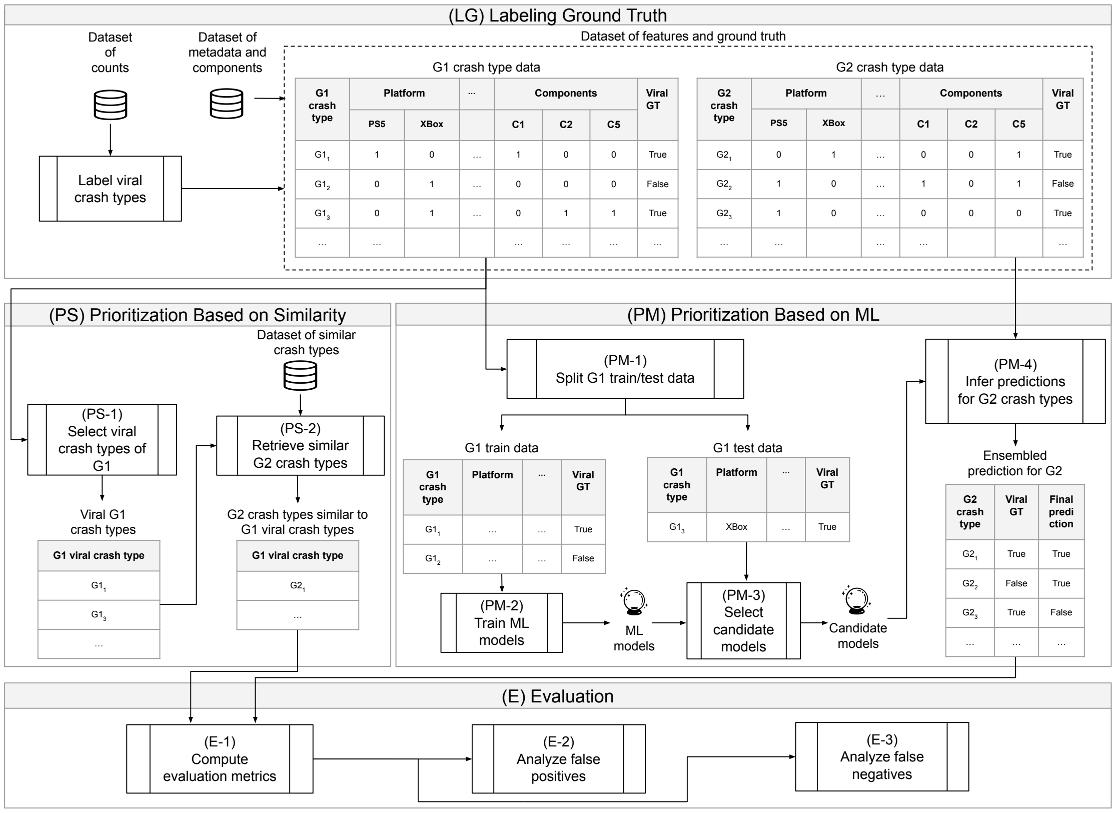
(LG) Labeling Ground Truth

In this step, we retrieve ground-truth labels, i.e., whether crash types are considered viral in the post-release phase. Fig. 5 (LG) shows an overview of this phase.

Viral crash types can be determined by the total number of occurrences or by sudden surges in frequency (Sec. IV). Thus, for a crash type c of either G1 or G2, we compute the total number of crash occurrences of c reported in the post-release phase by live players (i.e., c_{total}), and the maximum number of occurrences reported in a day by live players (c_{daily}).

For G1, we set a threshold t^{G1}_{total} such that a crash type c^{G1} is considered viral if $c^{G1}_{total} \geq t^{G1}_{total}$. Similarly, we set a threshold t^{G1}_{daily} such that crash type c^{G1} is considered viral if $c^{G1}_{daily} \geq t^{G1}_{daily}$. The crash type is considered viral if either or both conditions are met.

We follow a similar approach to label viral crash types of G2. In particular, we select total and daily threshold values and label the crash types in G2 that exceed either or both of these thresholds. Note that the chosen thresholds differ between the two titles due to substantial differences in the number of live players.


(LG) Labeling Ground Truth
Dataset of counts
Dataset of metadata and components
Dataset of features and ground truth
Label viral crash types
G1 crash type data
G2 crash type data
(PS) Prioritization Based on Similarity
Dataset of similar crash types
(PS-1) Select viral crash types of G1
(PS-2) Retrieve similar G2 crash types
Viral G1 crash types
G2 crash types similar to G1 viral crash types
(PM) Prioritization Based on ML
(PM-1) Split G1 train/test data
G1 train data
G1 test data
(PM-2) Train ML models
ML models
(PM-3) Select candidate models
Candidate models
(PM-4) Infer predictions for G2 crash types
Ensembled prediction for G2
(E) Evaluation
(E-1) Compute evaluation metrics
(E-2) Analyze false positives
(E-3) Analyze false negatives

G1 crash type data

G1 crash type	Platform		...	Components			Viral GT
	PS5	XBox		C1	C2	C5	
$G1_1$	1	0	...	1	0	0	True
$G1_2$	0	1	...	0	0	0	False
$G1_3$	0	1	...	0	1	1	True
...	...		...	...	...	...	...

G2 crash type data

G2 crash type	Platform		...	Components			Viral GT
	PS5	XBox		C1	C2	C5	
$G2_1$	0	1	...	0	0	1	True
$G2_2$	1	0	...	1	0	1	False
$G2_3$	1	0	...	0	0	0	True
...			...	...			...

G1 viral crash type
$G1_1$
$G1_3$
...

G1 viral crash type
$G2_1$
...

G1 train data

G1 crash type	Platform	...	Viral GT
$G1_1$	...	...	True
$G1_2$	...	...	False

G1 test data

G1 crash type	Platform	...	Viral GT
$G1_3$	XBox	...	True

G2 crash type	Viral GT	Final prediction
$G2_1$	True	True
$G2_2$	False	True
$G2_3$	True	False
...	...	...

Fig. 5: An overview of the process of obtaining suggestions for G2.

We select threshold values in collaboration with the operations team at Ubisoft. After labeling the viral crash types in the two titles, we include these labels in our dataset of metadata and component features.

(PS) Prioritization Based on Similarity

In this phase, we use the similarity between G1 and G2 crash types to predict potential viral crash types of G2. Fig. 5 (PS) shows an overview of this similarity-based prioritization, which we detail below.

(PS-1) Select viral crash types of G1. Using the virality labels of Sec. V-A (LG), we select all the viral crash types in the G1 dataset.

(PS-2) Retrieve similar G2 crash types. For each viral crash type in G1, we obtain the similar crash types in G2 from our dataset of similar crash types that we obtained in Sec. III (SC). We consider these matches to be our similarity-based suggestions, which are likely to go viral in the post-release phase of G2. Suppose that the viral crash types in G1 are $G1_1$ and $G1_3$, and we find that $G1_1$ has a similar G2 crash type, which is $G2_1$. In this case, prioritization based on similarity, i.e., our similarity-based approach, recommends prioritizing $G2_1$, since it is likely a viral crash type of G2.

(PM) Prioritization Based on ML

In this phase, we use crash types of G1 to train ML models to predict viral crash types of G2 as shown in Fig. 5 (PM).

(PM-1) Split G1 post-release train/test data. We begin by randomly splitting the dataset of G1 into training (80%) and test (20%) corpora. We use the training corpus to train ML models in Sec. V-A (PM-2), and the test corpus to select candidate ML models in Sec. V-A (PM-3) for evaluation on the G2 dataset in our Evaluation (E) phase.

(PM-2) Train ML models. We use our training corpus of G1 crash types to train ML models under various settings:

1) **Class-balancing settings.** Our dataset is highly imbalanced, with the viral crash types in G1 constituting only 0.6% of the crash types in G1. To address this, we apply random oversampling and Synthetic Minority Oversampling Technique (SMOTE) [6] to the training corpus. For each technique, we explore various sampling strategies by oversampling the records from the minority class to proportion $p \in \{0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5\}$ of the number of crash types in the majority class. Doing so results in 16 (eight p settings and two oversampling techniques) variants of our training corpus.
2) **Machine-learners and hyperparameter settings.** For each oversampled variant of our training corpus, we train logistic regression, neural network, gradient boosting, random forest, and K-nearest neighbor classifiers. For each classifier, we explore multiple hyperparameter settings. A detailed list of these settings is provided in Table B.1 in our online appendix.[8] This step produces 576 classifier variants.

(PM-3) Select candidate models. Classifiers may have poor fitness. Such poor classifiers should not be making predictions for G2. Thus, we apply each classifier to our test corpus of G1 and compute the precision and recall. We then select the models that achieve non-zero precision and recall values. We purposefully select a lenient lower bound to retain as many models as possible to maximize the overall recall of our approach. Doing so results in 384 candidate models selected for making recommendations for G2.

(PM-4) Predict viral crash types in G2. We feed the features of crash types in G2 to each candidate model, and use a voting-based ensemble recommendation. Our approach considers a crash type c^{G2} to be viral if the majority of candidate models predict c^{G2} to be viral.

(E) Evaluation

In this phase, we evaluate our approaches to predict viral crash types in G2. Besides our similarity-based approach and ML-based approach, we introduce combined approaches for evaluation, which label a crash type c^{G2} as viral if:

1) (CA1) both similarity-based and ML-based approaches predict c^{G2} as viral.
2) (CA2) either the similarity-based or ML-based approach predicts c^{G2} as viral.

(E-1) Compute evaluation metrics. To evaluate our approaches, we use precision, recall, and the following metrics:

1) **Area Under the Receiver Operating Characteristics Curve (AUROC) [12].** This metric estimates the discriminatory power of a classifier; AUROC values of 0, 0.5, and 1 represent the worst discrimination, random guessing (i.e., baseline performance), and perfect discrimination, respectively.
2) **Area Under the Precision and Recall Curve (AUPRC).** This measures the classifier's effectiveness in handling the minority class (i.e., viral crash types). The AUPRC is also a value between 0–1. The baseline performance of AUPRC is determined by the prevalence of the positive class, i.e., $\frac{\textit{number of viral crash types}}{\textit{number of all crash types}}$ [33]. While the baseline appropriate for a balanced-class distribution is 0.5, the baseline for our dataset of G2 crash types is 0.02.
3) **Matthew's Correlation Coefficient (MCC) [23].** This metric evaluates classifier performance to balance precision and recall. An MCC value of 1 indicates perfect classification, 0 indicates random guessing, and values below 0 reflect misclassification, with -1 representing total misclassification.

(E-2) Analyze false positives. Minimizing the number of false alarms is essential to effectively prioritize developer time for fixing the most impactful crash types. To assess the impact of false alarms, we focus on the prioritization approach that outperforms the others in evaluation metrics discussed in Sec. V-A (E-1), and examine the extent to which the total number of crash occurrences reported as false alarms deviates from the threshold (t^{G2}_{total}).

(E-3) Analyze false negatives. Our approaches can misclassify viral crash types as non-viral, resulting in false negatives. To understand the implications of any missed viral crash types, we compare the impact of correctly classified and misclassified crash types using statistical analysis.

We first determine whether the viral crash types of G2 that are misclassified are more impactful than those that are detected. To do so, we conduct a Mann-Whitney U test [22], comparing the total number of crash occurrences (c^{G2}_{total}) per crash type c between the two groups. Mann-Whitney U test is a non-parametric test that is used to assess the likelihood of two samples being drawn from the same distribution. Our null hypothesis H_{total} for the Mann-Whitney U test (unpaired, one-tailed, α = 0.05) is as follows:

(H_{total}) Null hypothesis for total number of crash occurrences

The total number of crash occurrences of correctly classified viral crash types is less than or equal to those of misclassified viral crash types.

Next, we use the Mann-Whitney U test (unpaired, one-tailed, α = 0.05) to compare the maximum number of daily occurrences (i.e., c^{G2}_{daily}), with the following null hypothesis H_{daily}:

(H_{daily}) Null hypothesis for maximum daily crash occurrences

The maximum number of daily crash occurrences of correctly classified viral crash types is less than or equal to those of misclassified viral crash types.

B. Results

Table I shows an overview of the evaluation of the predictions for G2, and below, we discuss our observations.

Observation 2: Our prioritization approaches outperform AUROC, AUPRC, and MCC baselines. Table I shows that the AUROC values for all four prioritization approaches exceed the baseline performance of the random guesser, which has an AUROC of 0.5. Similarly, the AUPRC for our similarity-based prioritization approach and combined prioritization approaches (CA1 and CA2) substantially surpass the baseline AUPRC of 0.02, showing that our approaches are effective at prioritizing viral crash types. However, the ML-based prioritization narrowly outperforms the baseline. Besides, the MCC values for all approaches are substantially greater than those of a random guessing model with an MCC value of zero, indicating that our prioritization approaches have a better overall performance in handling the imbalanced data.

Observation 3: Our similarity-based prioritization approach outperforms other approaches. Table I shows that our similarity-based prioritization achieves the highest MCC value among the studied approaches, indicating an effective balance between precision and recall. This similarity-based prioritization also ranks second in terms of recall (0.53) and precision (0.44). Although the highest recall (0.54) is achieved by the combined approach CA2, its precision (0.29) is substantially lower than that of the similarity-based approach.

TABLE I: Evaluation of our predictions for G2 crash types.

Prioritization Approach	Precision	Recall	AUROC	AUPRC	MCC	Occurrences of viral crash types (%)
Similarity-based	0.44	0.53	0.76	0.24	0.47	40.37%
ML-based	0.20	0.19	0.59	0.05	0.17	14.95%
CA1	0.55	0.18	0.57	0.11	0.30	14.70%
CA2	0.29	0.54	0.76	0.17	0.38	40.62%

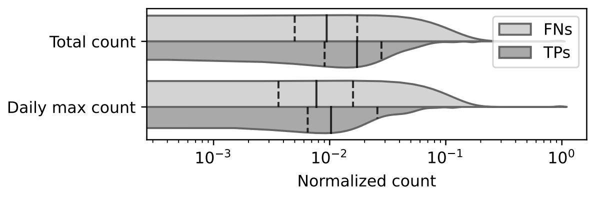


Fig. 6: Violin plot showing the distribution of the normalized number of (1) crash occurrences per G2 crash type and (2) maximum crash occurrences reported in a day per G2 crash type. X-axis is in a log scale. The solid lines represent the medians; the dotted lines represent the first and third quartiles.

Conversely, the highest precision (0.55) is achieved by our combined approach CA1, but it only achieves one-third of the recall of that of our similarity-based prioritization. Overall, similarity-based approach offers the best performance, striking a good balance between precision and recall in comparison to the others.

Observation 4: Our similarity-based prioritization identifies more than half of the crash types that grow virally in the post-release phase. Table I shows that our similarity-based prioritization successfully detects 53% of the viral crash types of G2 (recall = 0.53), demonstrating its effectiveness in identifying high-impact crash types. Furthermore, it captures 40.37% of the total crash occurrences attributed to viral crash types, indicating that this prioritization approach not only identifies a large number of individual viral crash types but also covers a substantial portion of the total volume of occurrences caused by viral crash types. This percentage of total crash occurrences attributed to viral crash types (40.37%) is less than the percentage of viral crash types that our similarity-based prioritization detects (53%) due to a false positive, which is an outlier crash type with a substantial number of occurrences. Note that our combined approach CA2 covers more crash occurrences than our similarity-based prioritization approach, but with considerably lower precision.

Observation 5: Although false positives of our similarity-based approach do not meet virality thresholds, 11% of them contribute over half the total occurrence threshold. Our similarity-based approach for detecting viral crash types achieves the lowest false alarm rate (0.56, i.e., $1 - precision$). However, our analysis of false positives reveals that 11% of the false alarms produced by our similarity-based approach accrue at least half of t^{G2}_{total}. Although they do not reach the threshold, their proximity to this threshold suggests that they are still worthy of prioritization.

Observation 6: Viral crash types that are detected by our similarity-based approach are more impactful than those that are missed as false negatives. Our statistical test to observe any differences between the total number of crash occurrences (c^{G2}_{total}) of correctly classified viral crash types and those of misclassified ones yields a p-value less than 0.05 with a non-negligible effect size (Cliff's $|\delta|$ = 0.29), indicating that the null hypothesis H_{total} can be rejected. Thus, the viral crash types of G2 that are correctly classified tend to have significantly more crash occurrences than the viral crash types that are misclassified.

Similarly, our statistical test for peak daily crash occurrences (c^{G2}_{daily}) of correctly classified viral crash types and those of misclassified ones yields in a p-value less than 0.05 with a non-negligible effect size (Cliff's $|\delta|$ = 0.22). Thus, we can reject the null hypothesis H_{daily}, and conclude that the viral crash types that are correctly classified in G2 tend to have significantly larger peak numbers of daily crash occurrences than the viral crash types that are misclassified.

Fig. 6 shows the normalized distribution of total crash occurrence counts and the maximum number of daily crash occurrences for correctly and incorrectly classified viral crash types. The median total crash occurrences for correctly classified viral crash types is nearly twice that of the incorrectly classified viral crash types. Moreover, the median of the peak daily crash occurrences for correctly classified viral crash types is 32% greater than that of the incorrectly classified viral crash types.

Our similarity-based prioritization (AUROC = 0.76) outperforms the ML-based prioritization (AUROC = 0.59) in detecting viral crash types, successfully identifying 53% of the crash types that grow virally during the post-release phase. Furthermore, the total and maximum daily crash occurrences for viral crash types that are correctly classified by our similarity-based prioritization are significantly larger than those of viral crash types that are misclassified.

VI. Threats to Validity

In this section, we describe the threats to the construct, internal, and external validity of our study.

A. Construct Validity

We rely on Ubisoft's crash reporter for deduplication of crash occurrences, forming crash types, and then use CodeLlama embeddings to compute the cosine similarity of the frames in the stack traces of these crash types. While the CodeLlama model may have been trained on general source code datasets, stack traces generated from the game-specific source code may contain game-specific nuances that the CodeLlama embeddings might not fully capture. As a result, crash types that appear similar based on the embeddings may actually be different. Nonetheless, the goal of using CodeLlama embeddings is not for deduplication (which is already handled by the organization's crash reporter), but is instead to find crash types with similar stack traces. We inspect a sample of 400 crash-type pairs with 99% similar stack traces and find them to be meaningfully similar, empirically strengthening confidence in the effectiveness of the embeddings for this task.

B. Internal Validity

In this study, we obtain viral crash types of each title, G1 and G2, by using thresholds for the total occurrences accrued over time (i.e., t_{total}) and the maximum occurrences reported per day (i.e., t_{daily}). The values set for these thresholds may introduce selection bias. To mitigate such bias, we validate these thresholds with Ubisoft's operations team.

Our dataset is highly imbalanced, i.e., viral crash types represent is 0.6% of the crash types reported in the post-release period of G1. To balance the classes in our training dataset, we use random oversampling and SMOTE methods. We choose not to use standard class balancing proportion, i.e., oversampling the minority class to match the number of records from the majority class, because while doing so decreases false negative rates, it also tends to increase false positives rates. To mitigate the bias from the proportion we choose for our oversampling methods, we train separate ML models for a range of proportions.

The choice of hyperparameters in our ML models may have an impact on our results. While we mitigate this by training ML models under several hyperparameter settings, the possibility remains that other hyperparameter configurations—that are not explored in this study—could yield different results. To promote replicability, we provide the hyperparameters used in this study in our online appendix.[8]

C. External Validity

Our analysis is based on two titles published by Ubisoft. The features extracted from crash types in these titles may differ from those extracted from titles developed using different game engines. Furthermore, the findings of this study may not be generalizable to titles published by other organizations. Despite these limitations, we posit that the proposed approach—leveraging crash reports from previous titles to predict high-impact crash types in new titles—could be adapted to other organizations and settings. Nonetheless, replication studies may strengthen the generalizations that can be drawn.

VII. Related Work

In this section, we situate our work with respect to the literature on software crashes and video game crashes.

A. Software Crashes

Several studies have proposed methods to warn users about software crashes in advance [1], [40], [44]. For example, Adam et al. [1] analyzed 5,598 user sessions of a medical application to identify combinations of user actions that led to crashes of a medical application. They used hierarchical clustering to define groups of sessions and computed the crashing probabilities for each cluster. During a working session, a crash probability was computed for each new user action based on the session assignment to predefined clusters. This method resulted in a specificity of 0.91 and a recall of 0.77.

Prior studies also analyzed software releases that crash applications [17], [43]. For example, Xia et al. [43] trained ML models on release-related data (e.g., number of file changes in a release) of ten open-source mobile applications that produced 2,638 releases. Their ML models achieved, on average, F1 and AUROC scores of 0.30, and 0.64, respectively.

Others proposed methods to localize crash-inducing defects in source code [26], [38], [41], [42], [46]. For example, Wu et al. [42] proposed CrashLocator—a method to locate faulty methods using the stack traces attached to crash reports. This tool ranks the methods in a stack trace by a suspiciousness score, which measures how often methods appear in the stack traces of crashes in comparison to other types of issues. They evaluated this approach on Mozilla crash reports and found that 50.6% of crashes are fixed by updating the top method in the stack trace.

The sheer volume of crash reports and the velocity at which they accumulate in popular software presents management challenges and can overwhelm organizations [5], [8], [25]. To manage this, crash-report deduplication techniques have been proposed to consolidate redundant crash reports [3], [7], [15], [28], [29], [32]. For example, Rodrigues et al. [29] introduced a crash report deduplication technique based on stack traces. Their approach outperforms previous deduplication approaches for Ubuntu project by 4.20% in terms of AUROC. Note that the in-house crash reporting system at Ubisoft (Sec. II), from which the data for our study is extracted, deduplicates crash reports for each title published by the organization, and categorizes them into a set of crash types. We also use CodeLlama embeddings [31] to find similar crash types across titles that do not meet strict similarity criteria (e.g., exact stack trace matches).

While much of the aforementioned research focused on detecting the crash likelihood of software and investigating crash root causes, a key unaddressed concern is the prediction of crash impact. Understanding the impact is crucial for prioritizing crash types, especially in large software systems with a large volume of crash reports that accrue at a high velocity. Kim et al. [14] were the first to focus on prioritizing debugging efforts by predicting potentially prevalent crash types. They selected crash reports of the 20 most prevalent crash types and crash reports of the 20 least prevalent crash types in a Mozilla Firefox release. Then, they trained ML models on this dataset to predict whether a given crash report from a new Firefox release is likely to come from a prevalent crash type or not. Their models achieved an accuracy of 0.63–0.81. Then, they used their models trained on Mozilla Firefox to predict prevalent crash types in a new release of Mozilla Thunderbird, achieving an accuracy of 0.54–0.69. Although Firefox and Thunderbird are two different software, they share crash types.[9]

[9]https://support.mozilla.org/en-US/kb/thunderbird-crashes

Although the approach by Kim et al. [14] makes important contributions, it cannot be directly applied to Ubisoft's game development context. First, unlike Firefox and Thunderbird, the video games G1 and G2 that we study in this paper do not share crash types detected by the crash reporter. Second, the release cycles differ in that new versions of Firefox and Thunderbird are released frequently, whereas triple-A video games are released infrequently. In the video game industry, developers often work on a title for months or years, and release it as a large-scale scheduled launch, where the title has one official release. For a completely new title, data from live players is not available until the release of that title to anticipate which crash types could be prevalent. Lastly, our ML-based approach considers all crash types in the dataset rather than restricting the analysis to the top 20 most prevalent or least prevalent crash types. This approach enables the detection and prioritization of a broader range of crash types, including those that may not fall within the extreme tails of the prevalence distribution.

B. Video Game Crashes

Prior work has proposed ML and reinforcement learning (RL) methods to automate gameplay testing to detect bugs in games, such as crashes [2], [9], [20], [34], [47]. For example, Zheng et al. [47] proposed WUJI, a testing framework that uses RL to automate gameplay testing. Wuji aims to strike a balance between winning the game and exploring the space of the game. Winning the game allows the agent to progress through the game events, while space exploration increases the coverage of the test, and in turn, the likelihood of discovering crashes and other types of bugs.

Other studies [10], [19] proposed methods to detect bugs (including crashes) from gameplay videos. For example, Guglielmi et al. [10] proposed GELID, an automated approach to extract segments from gameplay videos that contain bugs and crashes. In particular, GELID uses video-based features, such as the Structural Similarity Index Measure (SSIM) [39], to detect frame anomalies that occur during a crash.

Others adopted the game developers' perspective on video game crashes [18], [36]. For example, Truelove et al. [36] surveyed game developers about their experience with different types of bugs. They found that game crashes are the most frequently occurring bugs and are often treated as more severe than other types of bugs. They also found that game developers face challenges in detecting root causes of game crashes. Moreover, they found that certain change sets that are applied to the source code of a video game can cause previously fixed crashes to reappear in the game, which is a common problem in software bugs [48].

Much of the previous research on video game crashes has focused on their detection. To the best of our knowledge, the prioritization of crash types based on the breadth of their anticipated impact on players has not yet been explored. Our study strives to fill this gap by examining patterns in the accrual of crash occurrences of crash types and proposing approaches to prioritize crash types.

VIII. CONCLUSION AND IMPLICATIONS

In this paper, we analyze crash types in video games. We find that while the majority of crash types induce few occurrences, a subset are viral crash types that propagate broadly across a large number of players. Leveraging these insights, we propose approaches to prioritize crash types based on their likelihood of being viral. Below, we outline the implications of our study.

Teams can anticipate viral crash types in a new title by leveraging data from crash types observed in previously released titles. We find that the prioritization approaches proposed in our study outperform the baseline models in detecting viral crash types. Depending on the context of the titles and the specific metrics that need to be optimized (e.g., precision or recall), teams can select an approach from the four approaches proposed in this study. For example, if a team seeks to achieve a balance between precision and recall, the similarity-based approach tends to perform best. Conversely, if the goal is to maximize precision, the combined approach CA1 would be a better choice. These prioritization approaches would not only enhance the gaming experience for players, but would also reduce the workload of operations teams by potentially preventing occurrences of crashes in the post-release phase.

Teams may use stack trace similarity to target faulty components for long-term improvement. We find that prioritizing crash types in a new title based on stack trace similarity outperforms ML approaches, successfully detecting over half of the viral crash types. These findings highlight the potential of using stack trace similarity not only for prioritizing crash types, but also for other applications. For instance, by grouping crash types based on stack trace similarity, teams can identify components implicated by the largest proportion of crash types. Targeting such faulty components for further improvement could enhance the resilience of both games and game engines.

Promising directions for future work include the formulation of additional features of crash types, such as in-game state. Such features may highlight patterns that lead to crash types that become prevalent after the release. In this paper, our ML-based prioritization uses metadata and stack traces to predict viral crash types, and it yields a precision of 0.20 and a recall of 0.19. Incorporating other factors, such as the in-game state and player actions at the time of the crash, may improve the precision and recall. This paper lays the groundwork for using stack trace similarity and ML to predict viral crash types, and we encourage researchers who have access to additional data to build on our work to improve the prioritization of crash types in video games and other contexts. We have made the scripts used to perform the analyses of this study available in our replication package.[8]

ACKNOWLEDGMENTS

This work was supported by Mitacs and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] C. Adam, A. Aliotti, and P.-H. Cournède, "Learning from user workflows for the characterization and prediction of software crashes," in *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2016, pp. 1023–1030.

[2] S. Agarwal, C. Herrmann, G. Wallner, and F. Beck, "Visualizing ai playtesting data of 2d side-scrolling games," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 572–575.

[3] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 183–192.

[4] I. F. Ashari, E. D. Nugroho, R. Baraku, I. N. Yanda, R. Liwardana *et al.*, "Analysis of elbow, silhouette, davies-bouldin, calinski-harabasz, and rand-index evaluation on k-means algorithm for classifying flood-affected areas in jakarta," *Journal of Applied Informatics and Computing*, vol. 7, no. 1, pp. 95–103, 2023.

[5] J. C. Campbell, E. A. Santos, and A. Hindle, "The unreasonable effectiveness of traditional information retrieval in crash report deduplication," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 269–280.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[7] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1084–1093.

[8] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 333–342.

[9] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén, "Improving playtesting coverage via curiosity driven reinforcement learning agents," in *2021 IEEE Conference on Games (CoG)*. IEEE, 2021, pp. 1–8.

[10] E. Guglielmi, S. Scalabrino, G. Bavota, and R. Oliveto, "Using gameplay videos for detecting issues in video games," *Empirical Software Engineering*, vol. 28, no. 6, p. 136, 2023.

[11] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[12] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve." *Radiology*, vol. 143, 1982.

[13] Z. Harris, "Distributional structure," 1954.

[14] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.

[15] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 486–493.

[16] V. Lakshmanan, S. Robinson, and M. Munn, *Machine learning design patterns*. O'Reilly Media, 2020.

[17] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko, "Characterizing the differences between pre-and post-release versions of software," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 716–725.

[18] D. Lin, C.-P. Bezemer, and A. E. Hassan, "Studying the urgent updates of popular games on the steam platform," *Empirical Software Engineering*, vol. 22, pp. 2095–2126, 2017.

[19] ——, "Identifying gameplay videos that exhibit bugs in computer games," *Empirical Software Engineering*, vol. 24, pp. 4006–4033, 2019.

[20] G. Liu, M. Cai, L. Zhao, T. Qin, A. Brown, J. Bischoff, and T.-Y. Liu, "Inspector: Pixel-based automated game testing via exploration, detection, and investigation," in *2022 IEEE Conference on Games (CoG)*. IEEE, 2022, pp. 237–244.

[21] A. Maiga, A. Hamou-Lhadj, M. Nayrolles, K. K. Sabor, and A. Larsson, "An empirical study on the handling of crash reports in a large software company: An experience report," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 342–351.

[22] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[23] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[24] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *Transactions On Software Engineering and Methodology (TOSEM)*, vol. 11, 2002.

[25] N. K. Nagwani, "Identification of duplicate bug reports in software bug repositories: a systematic review, challenges, and future scope," *Data Deduplication Approaches*, pp. 183–201, 2021.

[26] R. L. Nord, I. Ozkaya, E. J. Schwartz, F. Shull, and R. Kazman, "Can knowledge of technical debt help identify software vulnerabilities?" in *9th Workshop on Cyber Security Experimentation and Test (CSET 16)*, 2016.

[27] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking api protocol conformance with mined multi-object specifications," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 925–935.

[28] Y. Remil, A. Bendimerad, R. Mathonat, C. Raïssi, and M. Kaytoue, "Deeplsh: Deep locality-sensitive hash learning for fast and efficient near-duplicate crash report detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[29] I. M. Rodrigues, D. Aloise, and E. R. Fernandes, "Fast: A linear time stack trace alignment heuristic for crash report deduplication," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 549–560.

[30] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[31] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[32] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 499–510.

[33] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets," *PloS one*, vol. 10, 2015.

[34] A. Sestini, L. Gisslén, J. Bergdahl, K. Tollmar, and A. D. Bagdanov, "Automated gameplay testing and validation with curiosity-conditioned proximal trajectories," *IEEE Transactions on Games*, vol. 16, no. 1, pp. 113–126, 2022.

[35] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 187–198.

[36] A. Truelove, E. S. de Almeida, and I. Ahmed, "We'll fix it in post: What do bug fixes in video game update notes tell us?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 736–747.

[37] L. VI., "Binary codes capable of correcting deletions, insertions, and reversals," *InSoviet physics doklady 1966 Feb 10 (Vol. 10, No*, vol. 10, no. 8, pp. 707–710, 1966.

[38] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 247–256.

[39] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[40] C. Wimalasooriya, S. A. Licorish, D. A. da Costa, and S. G. MacDonell, "Just-in-time crash prediction for mobile apps," *Empirical Software Engineering*, vol. 29, no. 3, pp. 1–62, 2024.

[41] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," *Empirical Software Engineering*, vol. 23, pp. 2866–2900, 2018.

[42] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 204–214.

[43] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proceedings of the 10th*

ACM/IEEE international symposium on empirical software engineering and measurement, 2016, pp. 1–10.

[44] M. Yakhchi, J. Alonso, M. Fazeli, A. A. Bitaraf, and A. Patooqhy, "Neural network based approach for time to crash prediction to cope with software aging," *Journal of Systems Engineering and Electronics*, vol. 26, no. 2, pp. 407–414, 2015.

[45] D. Zagieboylo and K. A. Zaman, "Cost-efficient and reliable reporting of highly bursty video game crash data," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 201–212.

[46] K. Zhao, Z. Xu, M. Yan, T. Zhang, L. Xue, M. Fan, and J. Keung, "The impact of class imbalance techniques on crashing fault residence prediction models," *Empirical Software Engineering*, vol. 28, no. 2, p. 49, 2023.

[47] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.

[48] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1074–1083.