

CREPE: A Privacy-Enhanced Crash Reporting System

Kiavash Satvat

University of Illinois at Chicago
ksatva2@uic.edu

Mahshid Hosseini

University of Illinois at Chicago
mhosse4@uic.edu

Maliheh Shirvanian

Visa Research
mshirvan@visa.com

Nitesh Saxena

University of Alabama at Birmingham
saxena@uab.edu

ABSTRACT

Software crashes are nearly impossible to avoid. The reported crashes often contain useful information assisting developers in finding the root cause of the crash. However, crash reports may carry sensitive and private information about the users and their systems, which may be used by an attacker who has compromised the crash reporting system to violate the user's privacy and security. Besides, a single bug may trigger loads of identical reports which excessively consumes system resources and overwhelms application developers.

In this paper, we introduce CREPE, a security-concerned crash reporting solution, that effectively reduces the number of submitted crash reports to mitigate the security and privacy risk associated with the current implementation of the crash reporting system. Similar to the currently deployed systems, CREPE aggregates and categorizes the crashes based on their root cause. On top of that, the server marks the crash categories in which sufficient reports have been received as “saturated” and informs the clients periodically through software updates. On the client, CREPE engages the reporting application in categorizing each crash to only submit reports belonging to non-saturated categories. We evaluate CREPE using one year of data from Mozilla crash reporting system containing 38,834,383 reports of Firefox crashes. Our analysis suggests that we can significantly reduce the number of submitted reports by bucketing 100 most frequent crash signatures at the client. This helps to preserve the security and the privacy of a significant portion of users whose data has not been shared with the server due to the redundancy of their crash data with previously submitted reports.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Browser security**; **Software and application security**.

KEYWORDS

Crash Reporting, Crash Report Privacy, Crash Report Security, Browser Privacy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CODASPY '20, March 16–18, 2020, New Orleans, LA, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7107-0/20/03...\$15.00

<https://doi.org/10.1145/3374664.3375738>

ACM Reference Format:

Kiavash Satvat, Maliheh Shirvanian, Mahshid Hosseini, and Nitesh Saxena. 2020. CREPE: A Privacy-Enhanced Crash Reporting System. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20)*, March 16–18, 2020, New Orleans, LA, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3374664.3375738>

1 INTRODUCTION

Many software companies have switched to the prevalent industrial practice of Rapid Release Process which accelerates time to market. However, this agile process leaves less time for the quality testing, limiting it to only error-prone areas, and therefore, leads to more reliance on the Automatic Crash Reporting System (ACRS) to collect crash information from the clients and recognize errors that have not been noticed in the development stage.

However, such an approach will not be free of cost for the users. The collection of crash reports leaves a privacy burden on the users' shoulder by inevitably collecting user's data. While the collected crash information contains valuable data to detect the cause of the errors, it also contains users' private and sensitive information. It is a well-known fact that the memory dump (that is submitted as part of the crash reports to the servers) contains applications sensitive information such as usernames, passwords, and encryption keys. Several types of attacks, as well as forensic investigation methods, have been introduced in the past to extract the hidden information laid in the memory [29, 51, 54]. Apart from the memory dump, the collected runtime information about the users' system at the time of the crash (e.g., IP address, visited URL, and os version) can be used to reveal the identity of users and therefore compromise the users' privacy [25, 30]. The study presented by Satvat and Saxena [55] examined a dataset of browser crash reports and revealed numerous instances of leaked private data, jeopardizing users' privacy and security. The study reported a significant number of IP addresses, visited URLs, and other identifying information including the presence of over 20,000 access tokens and session ids, 600 passwords, 9,000 email addresses, and a vast number of contact information. Such information could be used by an attacker who gets access to the crash data to harm user's security and violate their privacy.

In addition, the majority of reports are redundant reports flooding the ACRS with a volume of data which practically is infeasible to be processed in a timely manner, devouring the system resources and requires a considerable amount of human effort to analyze crashes [28, 53]. In a majority of cases, a bug can be fixed with a few instances of a crash, and remaining reports are redundant data, generated due to the recurrence of the same bug on

different machines. Therefore, the redundant crashes, which form a considerable portion of reports, are superfluous of user's personal information. Firefox, for instance, collects around 2.5 million crash reports per week [4], while they are able to process only 10% of these reports and the remaining 90% are tagged as "duplicates" or "not critical," by fuzzy techniques or manually by the QA team [4, 6, 28].

Limited works that studied the privacy and security of crash reporting system [35, 36, 39], suffer from the potential impact on the service quality, including additional overhead and possible readability impact on the crash report (discussed in Sections 3 and 7). Therefore, despite all the efforts, crash reporting infrastructures are still extensively swamped with redundant reports that contain sensitive information, undermining user's privacy and security. A practical approach to address these concerns is to follow the notion of sharing the least but the most relevant information [52, 59]. The philosophy behind this practice is for the clients to avoid sharing the (highly likely sensitive) data that is not being processed at the server, due to the redundancy with the previously submitted reports. This mitigates privacy and security concerns and preserves a significant portion of users' privacy and security, minimizing the severity of the potential harm to the users if the system gets compromised or data gets leaked. In this paper, we propose CREPE, a crash reporting system, that engages the client-side crash reporting application in categorizing crashes and thereby minimizing the number of duplicate reports submitted to the server. Reducing the number of submitted reports benefits both users and software companies: 1) By retaining redundant reports, CREPE preserves the privacy and security of a significant fraction of users in case the server gets compromised, or data gets leaked (e.g., as reported [55]). 2) CREPE reduces storage and network overhead by minimizing the submission of redundant reports.

Contributions. The detailed contributions are as follows:

- **CREPE System Design (Section 4):** We propose CREPE in which the client can categorize each crash using a signature generation method similar to the one traditionally used by the server for crash bucketing. In line with the current crash reporting mechanisms, CREPE server aggregates and categorizes crashes based on crash signatures. Additionally, CREPE server marks the categories for which sufficient information has been received as "saturated" and frequently submits saturated list to the client.

CREPE client is equipped with a lightweight debugger that converts the raw stack trace data to a readable memory dump information to generate signatures from the stack trace frame's module name. CREPE client submits the full crash report, including memory dump and crash descriptive information (i.e., user description and URL) to the server only for non-saturated categories. For sufficiently received crashes (saturated list) the client only submits the general crash information (e.g., crash id, signature, crash time, and system runtime information), so that the server can maintain information about the frequency and number of crashes. The unsubmitted crash memory dump and descriptive information can be stored locally on the client for future recall by the server. To reduce the debugging overhead on

the client, CREPE may only generate signatures for top frequent crash categories.

- **CREPE Evaluation (Section 6):** To analyze the feasibility of our approach, we run CREPE to generate signatures on the client and show that by equipping the client with a lightweight debugger, the client can generate the same signatures like the ones generated on the server. Since loading all the debugging information on the client may seem overwhelming, we suggest generating signatures for only top crash categories. The result of our analysis on one year's worth of crash reports submitted to Mozilla suggests that by only generating signatures for the 100 most frequent crashes, we can achieve a reduction of 43% in the volume of the reports sent to the server by avoiding transfer of 50% of duplicate reports, and thereby, significantly improve the privacy of the users as well as enhancing the utilization of system resources. This number can further reach as high as 83%, averaged over 27 versions of Firefox, if clients retain 80% of the redundant reports listed on the top 100 signatures. The remaining 20%, submitted to the server, is still far higher than the current volume that is being processed by the developers. We also demonstrate that the signature generation on the client side does not impact the performance of the client since the CPU and memory overhead is insignificant.

Generalizability. Our test is based on the open source Breakpad, a crash reporting system, which is utilized by many companies, including Mozilla and Google [1, 9]. However, CREPE can be integrated with any application as long as it uses the signature or any other unique identifier for clustering. To show the practicality of our system, we tested CREPE using real-world Firefox crash reports published by Mozilla [18], as the only publicly available source of the real-world crashes. We further expanded our experiment on Thunderbird to show that our work can be employed and integrated with other applications (Sections 6 and A).

2 BACKGROUND

Windows [10], Apple [11], Google Chrome [12], and Mozilla [19] are some of the major companies which extensively use ACRS to fix possible application failures and to improve the software quality. Despite the differences, they all use the same method for creating a unique signature to identify crashes. We based our design on Google Breakpad [1, 9], as one of the most dominant ACRS used by Mozilla products and many other applications.

2.1 Automatic Crash Reporting System (ACRS)

ACRS is comprised of two main components, namely, the client side crash reporter application, Controller, and the server side crash processing system, Processor. The two components collaborate to handle crashes¹ on a client and aggregate it on the server. The two can pair with crash analysis and reporting system(s) to report the crashes to the developers for further analysis and fixing the corresponding bugs². Controller collects and store crashes. Controller communicates with the Processor to send crash information including the client's system runtime information (e.g. OS and browser version), crash descriptive data (i.e., user

¹We refer to the failure of software as crash which leads the unexpected termination.

²We refer to the fault in the software as bugs which leads to an application crash.

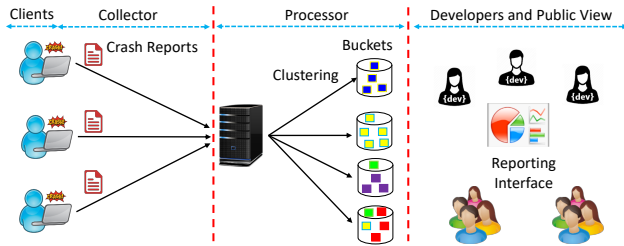


Figure 1: Overview of a crash reporting system architecture

description and URL), and the crash minidump. The Processor receives the crashes and aggregates them for further analysis by the developers.

2.1.1 Client Side (Collector). At the client side, the collector usually consists of an interactive interface which runs when the application terminates its normal execution process. The collector as an interacting interface between the Processor and clients, asks for users' permission to submit the crash report to the central crash repository or quit without sending the report. Choosing to send the report, the user still has the option to decide whether to include her visited website during the crash or not. The user also can share details of the issue which caused the crash through a description box. Additionally, there is an option to provide an email address for future support [2]. Figure 1 gives an overview of ACRS.

2.1.2 Server Side (Processor). Collected reports from the clients accumulate in a central crash repository for further analysis. The Processor is responsible for processing, analyzing, generating, and presenting reports to the developers. Each crash reporting system has an interface for presenting the collected data. Depending on ACRS configuration, the interface can be only used for troubleshooting by developers, or it can be accessible to the public. An example of the analyzed data that is available for public view is for the Mozilla crash reporting system [18].

2.2 Crash Bucketing

Field crashes are accumulated in a central repository and depending on the software can be colossal in number. Processing a huge number of reports can be a daunting task and can pose a significant overhead on the system's resources. As a result, companies need to have a proper clustering technique to group similar crashes into the same bucket for further processing. The bucketing helps the developers to deal with the crash in a more efficient manner. Bucketing algorithms use various features for clustering, e.g., [38, 40, 41]. In Firefox, Socorro clusters similar crash reports into the same bucket where each bucket is defined with a unique identifier called the crash signature describing the crash characteristics and identifying the bucket that each crash belongs to [3]. Signature generation algorithm is discussed more in Section 4.1.1.

2.3 Inside a Crash Report

Each Report contains a wealth of information about the state of a failure. While this information may be essential for debugging errors, it carries users' private data [15, 24, 36]. Each crash report generally carries two types of data. First, the minidump which is a stack trace of the failing thread, and consists of the sequence of

frames and functions in the memory at the time crash occurred. Second, the runtime information which includes the user's system data such as operating system and browser version, installed add-ons or plugins, and possible feedback provided by the user about the crash. In the case of browsers, the URL of the visited website at the time crash occurred is also collected as runtime information [24]. Having access to all this information enables the developers to investigate, replicate, and rectify the crash. Figure 2 is an abstract presentation of the crash data.

3 MOTIVATION AND RELATED WORK

3.1 Problem Statement

The current implementation of ACRS is a prevalent approach adopted by numerous software companies. However, this design has two major issues. The most severe concern with the present implementation of ACRS is the bulk of private information that converges from the users' systems into the crash report [13, 15]. Depending on the crash circumstances, each report contains sensitive information that jeopardizes users' security and privacy, ranging from PII to sensitive information such as username and password. For instance, the collected minidump, as a sensitive piece of data, may accommodate user's sensitive information such as username, password, and other data which was placed in the same memory space as the crash occurred [15, 24, 36]. Apart from the sensitive information that may appear in the minidump, crash runtime information, including URL of the visited website and users description, can turn each report into a piece of private data which can be traced back to a specific user, revealing information such as location, user's interests, and system status. Prior studies have reported on the practicality of such attacks using relatively less amount of information, compared to the data that currently resides inside crash runtime information [16, 34, 43]. For instance, URL can compromise user privacy by revealing the user's interest. In some circumstances (e.g., improper use of GET method) the URL may also carry sensitive information such as username, password, and details about sessions ids and access tokens. The description provided by users may also contain sensitive information such as contact details or other private data. The work of [55] reported many instances of private data appeared inside crash reports, including username and password which were shared over the description field, by non-expert users, aiming to receive further support.

The second drawback discussed in the previous studies [38, 47, 53], is the enormous amount of duplicate data that the current approach generates. This poses a considerable workload to the triaging team who at the very end is responsible for clustering crashes. This overhead, apart from utilizing human resources, can consume a massive amount of system resources including storage and network bandwidth. Ahmed et al. [28] mentioned the huge cost of crash reports and relatively small number of bugs that are eventually being identified – average 89 bug reports generated out of 96 million.

3.2 Prior Works

Broadwell et al. [36] and Castro et al. [39], studied ACRS from the privacy angle. They both tried to safeguard the user's private data by removing sensitive information from the dump. The approach

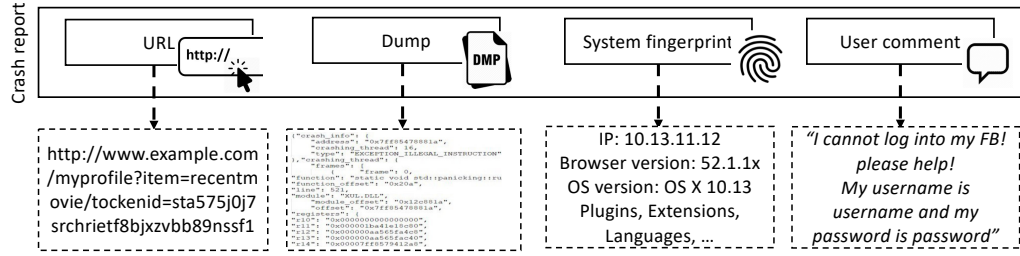


Figure 2: An abstract illustration of data residing inside a sample crash report

reads and compares the memory dump against a predefined list of sensitive data and tries to remove them from the dump. This approach, however, has several drawbacks. 1) Defining the sensitive list due to the linguistic characteristic of private data and the limitless ways that the private data can appear in the dump (e.g., languages and abbreviations) can be extremely complicated. Therefore, there is no guarantee that the right private data gets removed from the dump. 2) Transforming dump into a new form may cause additional overhead for the developers to read and debug the crash. Also, always there is a chance that removing data from the dump impacts the readability of the report and turns the dump into an unusable chunk of data. This probable readability impact becomes a major hurdle for the adoption of such practices in real-world scenarios. To the best of our knowledge, none of the suggested sanitization approaches has been adopted in the real-world ACRS systems due to this potential readability impact. 3) This approach, in the real world, can impact users' experience as it requires to inspect the memory dump and could take a considerable amount of time depending on the application structure, dump size, and order of the frames. This time is up to 100 seconds in [39], and up to 373% increase over the baseline in [36]. However, in our approach, generating a signature at the client is almost impalpable (discussed in Section 6.2). 4) Adoption of such approaches on a complex and relatively big software can be challenging, as the presence of an infinite number of constrained and unconstrained factors such as underlying operating system, associated software, and add-ons make the detection of sensitive data inside the memory dump extremely daunting and almost impossible in a feasible time. 5) None of these studies considered the presence of sensitive data in the crash runtime information, including private data that may appear in the user description and URL. Unlike these approaches, CREPE proportionally protects users' sensitive data present in the descriptive part of the crash report. Table 1 summarizes the differences between CREPE and the other approaches.

On the other hand, the studies conducted to enhance the efficiency of ACRS mainly applied various machine learning techniques to improve the bucketing and triaging of the crashes [32, 40, 48, 49]. The main motivation for most of the previous studies was accelerating and improving the system on the server side, by reducing the system's overhead using re-bucketing and detecting the duplicate crashes algorithms and techniques. However, these approaches have two main problems. First, applying these techniques still may pose an extra computational load on the system. Second, approaching the subject from the server side cannot result in any improvement in preserving the users' privacy.

Relatively similar to our approach, Windows Error Reporting system (WER) utilizes a progressive approach to collect system errors [26, 46]. However, unlike CREPE which works at the application layer, WER by having access to the kernel, works at the OS level. Unlike WER, CREPE is a platform-independent application tested on both Linux and Windows. Moreover, applications that utilize WER service are forced to share the users' private data with a third party (i.e., Windows). Employing WER also requires substantial changes in the bucketing methodology which a company is using.

In contrast to previous studies, we attempt to enhance ACRS efficiency in two different aspects, user's privacy, and system utilization. To this end, unlike the other studies, we mainly focus on the client side. We employ a clustering mechanism in the client to reduce the number of crash reports submitted to the server. Using this proposed system, the server can still follow its same bucketing approach and a new bucketing feature will be added to the client. In other words, our system does not require any major changes at the server side.

4 CREPE DESIGN

System Model. Similar to the current ACRS, we assume the server to be trusted and sensitive information (e.g., full access to the

Table 1: Comparison between CREPE and prior approaches on enhancing users' privacy in ACRS. The full-filled circle (●) indicates a system that has the property or feature listed on the left of the table. The system that lacks a feature or property is signified by ○. The half-filled circle ◐ refers to the situation where the approach can only eliminate instances where the data in minidump matches a predefined list of sensitive data. The half-filled circle ◑ refers to the situation where our approach preserves the privacy of a fraction of users by not submitting redundant reports.

		Current Deployment	Castro et al. [39]	Scrash [36]	CREPE
Memory Dump	Protect PII	○	○	○	●
	Protect sensitive information	○	◐	◐	◑
Runtime Information	Protect PII	○	○	○	●
	Protect sensitive information	○	○	○	◑
Potential Impacts	No report manipulation	●	○	○	●
	Reduce system overhead	○	○	○	●
	No impact on user-system interaction	●	○	●	●

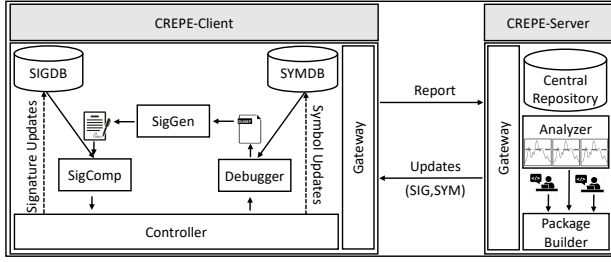


Figure 3: High-level architectural view of CREPE

memory dump) to be only accessible to trusted ones with restricted privileges. Given such a setting, we would like to limit the number of duplicate crashes submitted to the server to reduce the possible harms to the users' security privacy if the crash report data gets leaked to unauthorized users/attackers (i.e., as previously reported [55]). Our approach also attempts to use resources more efficiently. **Design Overview.** Our system encompasses two main components as shown in Figure 3: the client side crash reporter application, CREPE-Client, and the server side crash processing system, CREPE-Server. CREPE-Client sends crashes to CREPE-Server which aggregates and processes reports to resolve software failures. However, unlike the current systems in which only the server processes and categorizes crashes, in our design, both CREPE-Client and CREPE-Server can categorize crashes. Bucketing in CREPE-Client is a primitive clustering mechanism deployed by generating crash signatures, while equally primitive or more advanced bucketing techniques (e.g. [41, 49]) may be deployed at CREPE-Server. CREPE-Server can mark categories for which sufficient reports have been received as "saturated" and inform the client accordingly (based on the volume and frequency of crashes, and developers' needs for receiving more information on a specific crash). After categorizing the crash, CREPE-Client decides whether to include the crash descriptive data and dump with the report or not by referring to the list of saturated categories. Our crash reporting design can work on top of the current design (as it will be shown in Section 6 for Firefox and Mozilla crash reporting) with slight modification on the server side to mark saturated categories, and changes on the client side crash reporter to accommodate crash categorization.

4.1 System Components

4.1.1 CREPE-Client. Figure 4 illustrates the operational diagram of CREPE. The idea behind our design is for the client to categorize the crash by generating a crash signature derived from the system environment and a readable version of the stack trace. *Controller*, as the main component of CREPE-Client, receives a crash and sends the stack trace to *Debugger* to produce a human-readable dump from which the signature can be generated. Having the stack trace, *Debugger* generates a readable dump with reference to a debugging symbol file (*SYMDB*) loaded on CREPE-Client. The readable dump is then sent back to *Controller* where it will be aggregated with the environmental information and sent to the *SigGen* module. *SigGen* generates a signature to identify the bucket that the crash belongs to and sends it to *Controller*. This signature matches the one that is generated on the server. *Controller* sends a query $Q(SIG)$ to the local saturated signature database engine (*SIGDB*) through the signature comparator component, *SigComp*, asking whether it is

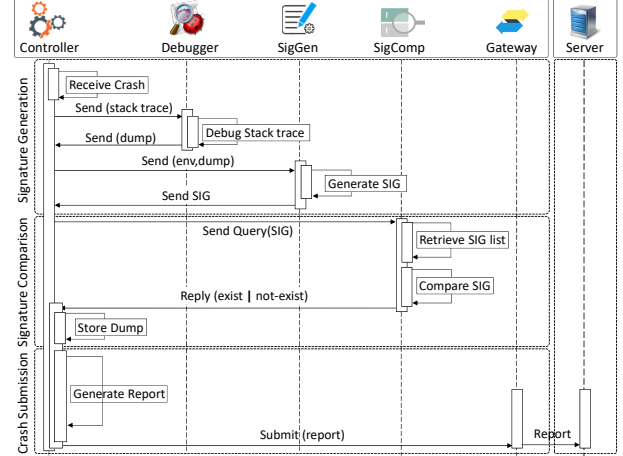


Figure 4: Overview of CREPE-Client component interactions

necessary to send the dump and runtime information to the server. *SigComp* looks up the latest saturated list and responds *Controller* whether to send the dump and runtime information to the server or not. If the queried signature does not exist in the saturated list this information should be sent to the server. In this case, CREPE-Client sends the full report to the server and keeps the crash id for further follow-up. If the queried signature exists in the saturated list, the client can store the dump and descriptive information in its internal storage and only submit the general crash information along with the signature to the server. The detailed design of the CREPE components is as follows.

Databases. To handle a crash at the client, CREPE-Client requires to have access to two types of data stored in local lightweight SQLite databases: 1) the list of saturated signatures, "*SIGDB*", and 2) symbol database, "*SYMDB*", containing the top crashes debugging information needed to create a readable dump from the stack trace. Both databases receive periodical updates from CREPE-Server through software updates. Storing the symbol files on the local machine may seem storage demanding. Considering that certain libraries and executables are often the main reason for the majority of crashes [27, 31, 45], loading the popular symbol files into the client is sufficient to decode the majority of crashes and generate signatures for top crashes (as will be discussed more in Section 6.1).

Debugger. Generating the signature at the client requires access to the stack trace and system runtime information. Most of the signatures are generated based on the stack trace of the crashing_thread. Others though, can be generated from the system runtime or the information available in the stack trace non_crashing_threads. Stack trace frame's signature is derived from the function names available in the readable dump. Unlike the server that has access to the debugging information, the currently deployed browser builds typically do not carry the debugging information. To address this, our CREPE-Client includes a debugger and symbol files (*SYMDB*). With access to the symbol file, *Debugger* creates a readable dump which is fed to the signature generator.

Developing a debugger or integrating it in CREPE-Client is out of the scope of this paper. A lightweight debugging tool called *minidump_stackwalk* has already been developed [8, 13] for debugging the stack trace. The *minidump_dump* tool built along *minidump_stackwalk* that can print the contents of the minidump.

Other tools such as `stackwalk_http` [8] are also available that can fetch the symbol file from Mozilla Symbol Server [21] and print the stack trace. Similar tools can be integrated into CREPE-Client to generate the crash signature from the stack trace.

SigGen. In our design CREPE-Client generates *SIG* based on the same rules defined and implemented by Mozilla Socorro [20]. Socorro creates the signature based on the stack trace crashing_threads frame's function name (i.e., function names). The signature generator reads the frames of the crashing_threads from the top of stack applying rules to the normalized frame's function name which are then concatenated to form the crashing signatures. Whether to include or not to include a frame name is decided based on Skip List rules [5] and Signatures Utilities Lists [22]. Most of the signatures can be generated in this way except for the following.

- **Abort:** This signature shows a controlled abort situation. This signature is created when the `AbortMessage` field is set in crash metadata. The signature is the "Abort" concatenated with the signatures of the other thread frames name following the same rules as generating normal signatures (i.e., the `crashing_thread` is not parsed but the signature is generated same as other `non_crashing_threads`).
- **IPCErrror-browser:** If the crash happens as a result of an Inter-Process Communication (IPC) error the generated crash is categorized as `IPCErrror`. If the `additional_minidumps` field in the metadata has the value of "browser" the signature is `IPCErrror-browser` concatenated with the value of the `ipc_channel_error` field in the metadata.
- **shutdownhang:** A hang during shutdown is categorized as `shutdownhang`. If the name of the top most stack frame contains `RunWatchDog`, the signature will be "shutdownhang" concatenated with signatures of the other thread's frames following the same rules as generating normal signatures.
- **OOM:** Out of memory errors are categorized as `OOM` concatenated with a size (i.e., small or large) determined by `OOMAllocationSize` field in the metadata. To detect these errors, the frame signature should be `CrashAtUnhandleableOOM`, `NS_ABORT_OOM`, `mozalloc_handle_oom`, or `AutoEnterOOMUnsafeRegion`.

SigComp. As a comparison function interacts with the *SIGDB* to check whether the generated signature already exists in the *SIGDB* or not. If the crash exists in the saturated list, *SigComp* flags the report as saturated, and informs Collector on this flagged report to keep the dump and descriptive data and only send the general crash information to CREPE-Server.

Gateway. This intermediary component interacts between the CREPE-Server and CREPE-Client. Once the signature is created, *Gateway* transfers the crash to CREPE-Server. *Gateway* also receives updates including symbol file and saturated signatures.

Controller. As the main component, *Controller* receives the crash, interacts with other CREPE-Client components, and prepares the report to be submitted to the server. Once the crash happens, CREPE-Client asks for the user's permission to submit the crash. If the user agrees to send the crash, *Controller* can run as a background process (to minimize any performance overload) while the browser reboots and the user proceeds to the normal operation. If the report is flagged as saturated by *SigComp*, *Controller* does

not send the descriptive information and memory dump to CREPE-Server and instead stores them in local storage. For this type of crashes, *Controller* generates and sends a report including crash ID for tracking, and general crash information (e.g., signature, crash time, browser version, and system environmental variable) which would be used for the statistical purposes and future references. For crashes on the non-saturated category, a full crash report will be submitted to the server.

4.1.2 CREPE-Server. Similar to the current crash reporting servers, CREPE-Server aggregates, buckets, and processes reports. Additionally, CREPE-Server is responsible for marking saturated categories and updating CREPE-Client databases. Our design is based on Socorro signature generation scheme, and we expect the servers to be able to generate the signatures and bucket crashes in a similar fashion. This approach is one of the simplest forms of bucketing that considers the crash in isolation and does not require information about other crashes to generate the signature. Since we require the client to also bucket the crash this is a feasible approach on the client side. However, the server may deploy other advanced techniques for more granular bucketing.

To mark saturated categories, CREPE-Server keeps track of the volume of crashes received in each category. Apart from the number of crashes, CREPE-Server considers other factors such as frequency of a crash, the severity of the bug that caused the crash, and developer's comments, to mark a crash category. The saturated list is shared with CREPE-Client frequently through a new version release. The main components of CREPE-Server are as follows.

Central repository. CREPE-Server accumulates, buckets, and analyses the reports in a central repository. It can also provide statistical reports for developers or public access.

Analyzer. As the main component of CREPE-Server, decides on the crash categories that should be in the saturated list. Having access to the threshold, crash frequency, and developer comments (whether more information for a crash type is needed or not), *Analyzer* decides to add or remove a signature from the saturated list. The result of this decision appears in the updates that CREPE-Client receives from CREPE-Server. Hence, if *Analyzer* decides to add a signature to the saturated list, CREPE-Server informs CREPE-Client by sending the updates to *SYMDB* to debug the crash and to *SIGDB* to determine if the dump should be sent to CREPE-Server.

To decide which crashes should be on the list, *Analyzer* can use a predefined threshold, heuristics algorithms, or the system administrators can add the crash signature into a saturated list – which contains signatures that have already reached the sufficient number of reports to detect and fix a bug. Therefore, any report received afterward is, perhaps, not required for fixing the bug. The saturated list can also contain signatures for crashes for which the root cause have been recognized and more reports is not required.

Package Builder. According to the analyzer outcome and developers' comments, CREPE-Server builds an update package for the client, including updates for *SIGDB* and *SYMDB* (retrieved from a centralized symbol server such as Mozilla Symbol Server [21] or Microsoft Symbol Server [17]).

Gateway. CREPE-Server receives the crash reports submitted by CREPE-Client through a communication gateway component and transfers it to the corresponding modules for processing. The update

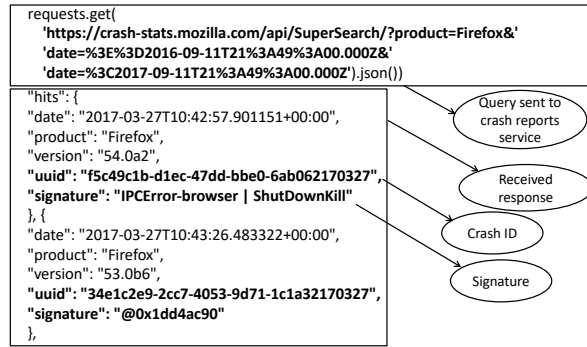


Figure 5: Query and response sent through Super Search

package can be transferred to the client (for example as a part of the browser’s update or as a separate crash reporter’s update) through the communication gateway.

5 CREPE IMPLEMENTATION

To show the feasibility of generating the signature at the client side, we developed CREPE using python to generate and compare the signature with the saturated list. We used the Mozilla crash reporting website [18] through which Mozilla presents crashes after processing them. Each crash report is assigned a crash ID and contains information received from the raw crash report and processed crash report. Most of the crash report fields are visible except for raw minidump that is available only to those with minidump access. Crash reports are presented in several tabs on the Mozilla crash reporting website, including Details, Metadata, Modules, Raw Dump, Extensions, and Correlations. Signatures can essentially be generated from the Raw Dump and Metadata field only corresponding to stack trace and crash information on the client side. Since we could not replicate all the crashes on the client side, we used Raw dump and Metadata to generate the crash signatures on the client side. This experiment serves to prove that signatures can be generated on the client side given that human-readable dump and metadata is available, which we already argued that they would be available on the client side in our design. The same approach has been taken in other works [38, 41, 56, 58] to evaluate bucketing schemes.

We developed CREPE in Python to retrieve the crash data from the Mozilla website, parse the data, generate signatures on the client side and compare them with the signatures generated on the website. If the generated signature matches with its pair from the website, this shows that the same crashes can be generated on the client and server. Our program uses Mozilla Super Search API [23] to fetch the data from the website. We send an HTTP request to the Mozilla crash-stats service, which returns a JSON document in response. The root of the JSON document contains hits key. hits key accommodates the crash reports matching the query and includes uuid (representing Crash ID), date, signature, product, and version fields. We only use the first two fields (uuid and signature) in our program. signature serves as the server side generated signature and uuid is used to retrieve Raw Dump for each Crash ID. Figure 5 shows a HTTP request and the received response from the super search.

The JSON document received from the crash-stats does not contain Raw Dump required for each Crash ID. To receive

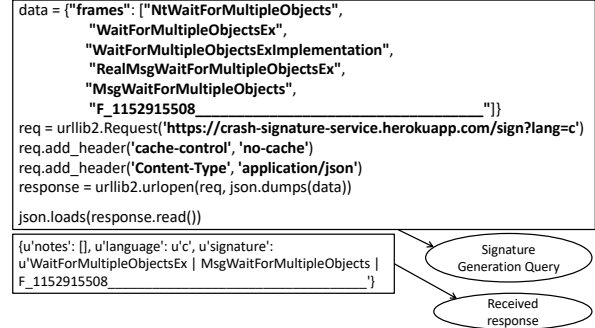


Figure 6: Signature Generation Service query and response

Raw Dump for a crash ID to create signatures, we scraped the webpage containing the Raw Dump for each Crash ID using BeautifulSoup library and extracted “rawdump” section from the webpage. We loaded this data into a JSON element and read all the Crashing Thread frames’ function name (i.e., [“crashing_thread”][“frames”][“function”]). As mentioned in Section 4.1.1, most of the signatures can be generated by concatenating frames’ function name and applying Socorro signature generation rules. To generate the signature based on the function names, we used and set up Crash Signature Service [7] on a client. Crash Signature Service [7] as a light weight signature generator builds the signature based on the stack trace frame’s signature. We installed the service on a local Linux machine and updated siglists by loading the latest siglist from Socorro Signature Utilities Lists [22]. Figure 6 shows a query sent to the Signature Generation Service and the received response.

For crashes of type OOM, abort, and IPCErrror-browser the signature can be generated on the client side from the Metadata only without access to the crashing thread in the stack trace. Therefore, it is trivial that the client can generate them locally even if the crash reporter does not have a debugger. For crashes of type shutdownhang and hang the signature can be generated from the Metadata (to define the crash type) and other non_crashing_threads in the stack trace in the same way that other crashes are generated (concatenating the frame signatures). The implemented algorithms at the server and client are represented by Algorithms 1 and 2 in the Appendix.

6 CREPE EVALUATION

Two central factors needed to be followed for a possible ACRS solution to be considered as an efficient and practical approach. The proposed mechanism should not impact the service quality both in terms of the process of fixing bugs and the users’ experience. To show that our system does not impact the process of fixing bugs, we demonstrate that there is a potential for reducing the number of crashes submitted to the server without causing harm to the process. To this end, we performed statistical analysis (Section 6.1) on various distributions of Firefox and showed that there is a significant similarity between crashes in Firefox versions which can be used to avoid sending further data that is not necessary for replicating and rectifying a bug. We build our main analysis based on Firefox crashes due to the popularity of this application and the significant number of publicly available crashes that can help to make the analysis more tangible. We further expanded

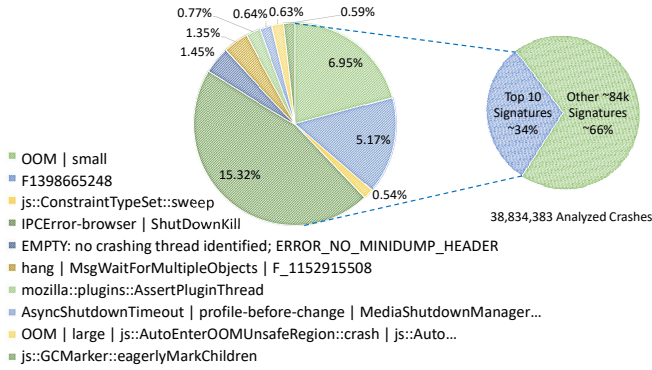


Figure 7: Distribution of the studied signatures and the top 10 frequent signatures from September 2016 to September 2017

our experiment on Thunderbird (Sections 5 and A) to demonstrate that such an approach can be adopted by other applications. We tested CREPE on both Windows and Linux to show that it leaves no overhead on the clients and has no impact on user’s interaction with the system (Section 6.2).

6.1 Analyzing Crash Volume Reduction

We study the real-world Firefox crash reports to show the potential to preserve a significant fraction of users’ privacy and reduce system overhead by not submitting the redundant reports.

Dataset. In our analysis, we considered crash reports submitted to Mozilla from September 2016 to September 2017. We analyzed 38,834,383 reports available on Mozilla crash reporting website [18] for all browser versions and platforms (i.e., Windows, Mac OS X, Linux, Others) and all process types (i.e., browser, plugin, content, gnu). We specifically analyzed reports of the 27 recent versions of Firefox (appears in Figure 8), including stable and beta versions to demonstrate the possible correlation between different versions.

Reducing the number of reports and generating signatures at the client may not be free of cost. The client needs to have access to a database of symbol files and conduct a process to generate the signature. Considering the huge number of signatures (84K), having all symbol files in the client can result in disk space issue and system slow down which in turn can have ominous impacts on system’s performance. To avoid such a cost, we put forward our hypotheses based on the fact that some parts of the software are more prone to produce bugs and errors [27, 31, 45]. Therefore, addressing these error-prone areas can result in a significant improvement by reducing the number of duplicates crashes. Therefore, our hypothesis is that there exists a similarity between application versions and more specifically successive versions, which helps the deployment of a system that significantly reduces the number of redundant crashes. Based on this fact and to confirm our hypothesis, we analyzed Firefox crash reports to find out if there exist any similarity between crashes in Firefox versions. We inspected this data in two different manners:

First, as an initial step, we studied one year data as a whole to validate our hypothesis and to understand which crashes are shared across all versions. During this period, we studied 38,834,383, and we noticed that close to 34% of crashes in all Firefox versions, including Nightly, stable, and beta are constituted from top 10

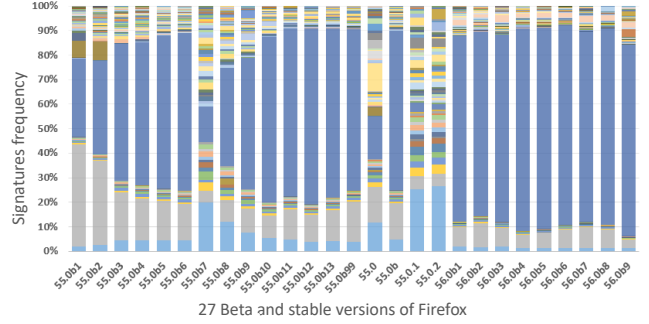


Figure 8: Distribution of the top frequent signatures between 27 versions of Firefox, where each color presents a signature.

signatures (Figure 7). In contrast, the remaining 66% constituted from approximately 84K signatures [6]. We further found out that among all the crashes, 16,578,398 belong to the top 50 frequent signatures (42.62%). While the obtained result potentially confirms the basis of our hypotheses (saying that addressing a minority of errors can result in a significant improvement in reducing the number of duplicates crashes), we took further steps to prove that the distribution density of prevalent crashes follows a pattern, and show that there is a close similarity across different versions.

Second, to prove the presence of similarity and correlation between different versions, we studied the frequency of the top 50 crashes across 27 versions of Firefox including core and beta versions. The initial result of our scrutiny has evoked the recurrence of the same pattern between different and successive versions. To confirm the presence of this similarity (the pattern can help developers on defining the saturated list), we performed a Wilcoxon Signed Rank Test [57] against our dataset. Using the Wilcoxon Signed Rank Test, we compared different and successive versions. The result of the test confirmed that in the majority of cases successive versions share relatively similar signatures. For instance, the result of Wilcoxon Signed Rank Test was $p = 0.964$ for two successive versions of 55.0b.5 and 55.0b.6; although, the differences exist between prior and later versions of 55.0.1 and 55.0.2 ($p=0.000$, $p=0.001$), due to the change from beta version to the core, yet they both share $p=0.877$ similarity in their top 50 crash signatures (appears in Figure 8). Similarly, although differences exist between 55.0b.7 and its prior and later versions, yet it shares significant similarity with stable versions like 55.0 ($p=0.177$), 55.0.1 ($p=0.391$) and 55.0.2 ($p=0.349$). Figure 8 demonstrates the pattern and the correspondence of top 50 signatures between different versions.

The obtained result confirms that similarity exists in the distribution of crashes across different Firefox versions, and suggests the potential for lessening the number of crash reports. To quantify this potential and provide some estimation, we analyzed the frequency of the top 100 signatures across different versions, to replicate the server requirement of having sufficient instances of a crash. We measured the occurrence of each crash across all versions independently and compared it with the total number of crashes to infer an estimation on the number of required crashes which needed to be submitted to the server. Our analysis was involved in comparing each signature frequency against a threshold to ensure that enough instances of a crash submitted to the server for debugging. Our initial assumption was that having access to 20% of the crash instances is sufficient for the process of debugging

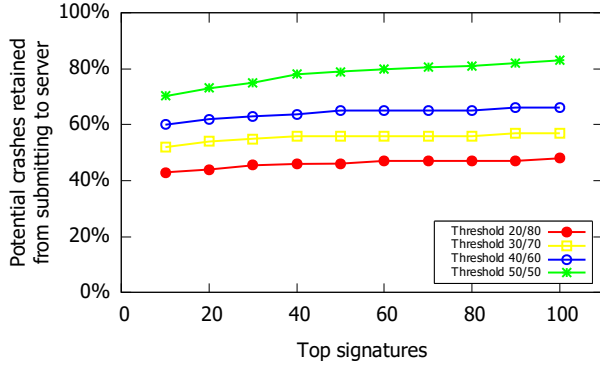


Figure 9: Correlation between the number of signatures generated at the client and potential reduction in submission of crashes by different thresholds

and fixing a bug. We set this number as a baseline based on the fact that only less than 10% of crashes are being used and proceed in the current implementation of ACRS, and the remaining are tagged as duplicates [4, 6, 28]. This number is still higher than the current number of instances that are being used to fix the bugs and is far beyond the developers' capability of processing crashes. However, we defined a minimum threshold of 20% to stay safe from not receiving sufficient instances of a crash. Note that in practice thresholds are defined by developers and can be set higher or lower than our assumed threshold.

Besides having the threshold of sending 20% of crashes and saving the remaining 80% from unessential submission, we expanded our analysis on the cases where the threshold is set higher, to send 30%, 40%, and 50% of the top 100 crashes and respectively save 70%, 60%, and 50% of them from submitting to the server. The results confirm that CREPE can reduce the number of crashes by 43% in the 50/50 threshold scheme by only processing of top 100 signatures at the client side. This number can further reach as high as 83% (averaged over 27 versions) in the 20/80 scheme if the client can categorize the most frequent 100 crashes. Figure 9 shows the correlation between the number of generated signatures and the potential reduction in the submission of crashes.

Sensitive Data Protection. Based on the CREPE implementation and the threshold scheme defined by developers, for each crash type, only a specific number of reports would be sent to the server and the remaining reports remain safeguarded at clients. However, not all crashes are equal in terms of carrying private information (discussed in Section 2.3). While some may only hold PII, the others may carry more sensitive data such as users' credentials. Therefore, in this section, we show the probability that the records containing the users' sensitive information would not be submitted to the server by CREPE. To this end, we assume that we have a total of N crash reports of which X contains users' sensitive data (e.g., username, password). CREPE will send only M crash reports to the server and keeps the remaining $(N - M)$ reports. Here is a set of assumptions: 1) We assume that each observation falls into just 1 of 2 categories: success (sensitive) and failure (non-sensitive). 2) Since, CREPE only sends a small portion of crash reports to the server, removing a few reports has a very small effect on the composition of the remaining population, so successive observations are very nearly independent. 3) We assume that all M reports that are being sent to the server have the same probability of success (sensitive)

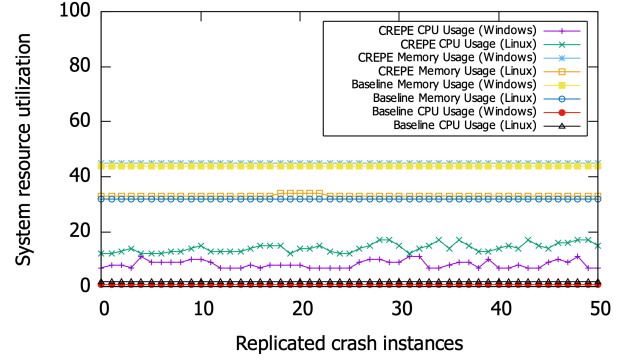


Figure 10: CREPE system utilization in comparison with baseline in both Linux and Windows platforms. y axis represents the utilization and x axis represents the aggregated instances of various crashes running on both platforms.

p . The population N contains a proportion p of successes, and we assumed that the population N is much larger than the sample size M . Therefore, the count X of successes in the simple random sample of size M has approximately the binomial distribution $B(M, p)$. The probability of getting exactly n successes (sensitive records) in M trials is given by the probability mass function:

$$\begin{aligned} Pr(n; M, p) &= Pr(X = n) \approx \\ C_n^M p^n (1-p)^{M-n} &= \binom{M}{n} \left(\frac{X}{N}\right)^n \left(1 - \frac{X}{N}\right)^{M-n} \\ C_1^M p^1 (1-p)^{M-1} &+ \\ C_2^M p^2 (1-p)^{M-2} &+ \\ C_3^M p^3 (1-p)^{M-3} &+ \\ \dots & \\ C_X^M p^X (1-p)^{M-X} &= \\ \sum_{i=1}^{i=X} C_i^M p^i (1-p)^{M-i} \end{aligned}$$

Therefore, the sensitive information will be saved from being submitted to the server by the probability of $1 - \sum_{i=1}^{i=X} C_i^M p^i (1-p)^{M-i}$. For instance, assuming that we have a total number of 40000 reports of a sample signature samplesignature where 2000 of these reports contain users' sensitive information. The probability of appearing 200 sensitive records in 8000 reports which are being submitted to server is $4.6755e-30$. This number is as low as $1.4050e-83$ if developers chose the 30/70 threshold to receive 12000 reports.

6.2 Performance Analysis

To demonstrate the feasibility of generating reports at the client, we generated and examined top signatures in various systems to measure timing and system resource utilization overhead. We tested both Firefox and Thunderbird on Windows and Linux platforms. Our Linux system was specifically chosen to represent low-end devices. Our 64-bit Windows (6.1, Build 7601) platform was run in 8.00 GB (7.87 Usable) RAM, and an Intel(R) core(TM)2quad q9550 @2.83GHz Processor. Our Linux platform was an Ubuntu x86_64 with 3902MiB System memory, and Intel(R) Core(TM)2 Duo CPU, T9600 @ 2.80GHz processor. We used python timing function to measure the latency caused by CREPE in computing the top crashes in the client. The average and the standard deviation of the delay in computing the signatures was $12.98ms(3.834ms)$ for the Windows platform and $10.864ms(2.201ms)$ for the Linux, averaged

over 1000 iterations of top crashes. Similarly, we measured the system resources in terms of memory and CPU to examine the possible impact of running CREPE at the client. We measured system utilization before, during, and after running CREPE. While we have not noticed any impact on client memory, the processor utilization varied between 7% to 11% in Windows platform and 12% to 19% in Ubuntu. The result is promising considering the system specification and the fact that no optimization performed on either platform or application level. Figure 10 shows the aggregated system utilization at the time of generating 50 sample signatures.

7 DISCUSSION

Saturated List. One central and influential factor in CREPE is the importance of defining an accurate saturated list. While an accurately defined list can increase the efficiency, adding a wrong signature to the list may impact the process of fixing bugs by not receiving sufficient instances of one crash type. To avert such a situation, we chose the top crash signatures for our analysis, since the abundant number of crashes in these categories helps to avoid crossing borderline of having or not having sufficient samples of a crash. But, similar to the current process that developers involvement is required for triaging and prioritizing bugs [28, 33, 50], our system relies on the developers to manage the saturated list.

Current Bucketing System. Ideally, the bucketing approach is expected to cluster crashes caused by the same bug into a distinct bucket. However, the real world implementation of the bucketing system is far from perfect [37, 38]. In some scenarios, a crash may map to more than one bug. A reason behind this is the imperfection of current signature generation algorithms which are not able to generate precise and detailed signatures that can perfectly match with the bug type and cause false mapping and the appearance of one-to-many and many-to-one relations between crash signatures and bugs. Having a more accurate and detailed signature generation scheme with the ability to bucket all similar crashes into the same exact group can result in saving more information from transferring and accordingly less effort for further triaging and processing. Therefore, any improvement in the current bucketing approach that results in more precise grouping helps improve our system and reduce the need of transferring data to the remote server.

While this is an orthogonal problem, the issue can be managed to a great extent for several reasons. First, the saturated list is defined and maintained by the developers for each version, meaning that a signature can be added to the list when developers ensure they have enough instances of that crash. This empowers developers to avoid adding controversial signatures to avoid the occurrence of false positive instances. Second, similar to the current push function which forces new updates prior to launching the application, a pull function can retrieve the un-submitted yet desired crash reports in cases where more information about a crash is required. Finally, such a concern is not applicable for many of crash signatures where the root cause is apparent. Instances which remain unfixed and repetitively appears in successive versions due to the unattended external factors like incompatibility issues of an extension or a plugin. Moreover, this issue is discussed as an addressable concern in [46], meaning that incorrectly diagnosing two report types as

having the same root cause result in the representation of the second report and dominating future reports after fixing the first error.

Other Defensive Mechanisms. Both server and client side approaches can be considered to enhance the security and privacy of users' information in ACRS. Although the server side approaches may provide more flexibility for the developers, helping them to have the best report readability, the user should not be obligated to trust the developer and the system to send their sensitive data to the server. Therefore, we focused on the client side approach to safeguard a significant fraction of the user's data. Other protective mechanisms can also be applied or integrated with our solution to provide additional layers of assurance.

In general, three types of approaches can be employed to secure data at the client. 1) The first solution is to mask or remove sensitive data in the report (e.g., work of [36] and [39]). This approach though protects users' privacy can impact the readability of the report. For instance, sanitization can turn the minidump to an unusable chunk of data or similarly removing data from the descriptive fields may impact the readability and may cause an additional overhead for the developers to debug the report. Such an issue is a serious impediment to employ this type of approach in the wild. 2) The second solution includes the use of techniques such as natural language processing or deep learning to build a model and protect user private data and avoiding the transfer of reports that are tagged as sensitive. Both approaches, however, are hard to implement since the probability for the appearance of private data is almost infinite. For example, username and password can appear in different crash fields in a variety of formats. This signifies the importance of labeling which is again hard, as it cannot be done using prevalent methods as they may jeopardize users' privacy. Moreover, testing the result of such approaches requires the manual check which again can result in violating users' privacy. 3) Finally, differential privacy and adding noise as another solution which recently been adopted in the wild. The method promises that the outcome of a survey remains the same with or without the attendance of a specific user [42]. However, such an approach may be hard to deploy in practice, the limited number of companies that adopted the technique, including Apple [14] and Google [44] limited their use to a certain context. For instance, Apple uses differential privacy on collected data to evaluate used images in a certain context. Google utilizes differential privacy in maps to collect traffic data in large cities. However, in the case of ACRS a variety of constrained and unconstrained factors such as diversity of platforms, versions, and also the possibility of appearing private data in diverse forms and various places are infinite.

Limitations and Future Directions. Although it was possible to generate the human-readable dump using a debugger on the client, we did not replicate all the crashes on the client machine, since crashes are highly dependent on parameters such as system hardware, OS, browser version, installed plugins, and visited URLs, making it impossible to replicate a crash. Besides, given that the crash raw data may contain sensitive information, crash datasets are not publicly available. Therefore, we relied on Mozilla dump data to generate signatures. Also, in the feasibility study (Section 6.1), we analyzed the distribution of the top 50 crash signatures between 27 versions of Mozilla Firefox across all platforms (i.e., Windows, Mac OS X, Linux, Others) and process types (i.e., browser, plugin,

content, gnu). However, a more detailed study on each platform or process type may lead to better categorization and therefore provide a better understanding of the crashes and signatures distribution.

As discussed in Section 1, we focused on Mozilla products as it offers access to the datasets of publicly available crashes, providing the ground for a real-world feasibility study. However, we believe the notion of submitting only the required amount of information is an intriguing one which could be usefully explored in further research, including employing such notion in areas of web and desktop applications. Further studies need to be carried out on mobile applications as another compelling line of research to explore the impacts of Android closed architecture and permissioning on deploying such a system in Android devices.

8 CONCLUSION

Against all defense measures, we are still witnessing an unprecedented amount of data leakages each year. One possible approach to address the current trend is to share the least possible amount of information. In this paper, we used this notion in the context of ACRS aiming to preserve users' privacy by reducing the number of duplicate reports collected from clients. We introduced CREPE that engages the client in the process of crash reporting. CREPE by adding a layer of bucketing at the client and utilizing the server to inform the client of the categories for which sufficient reports have been received prevents the submission of redundant reports and in turn preserves the privacy of a significant number of users. To study the feasibility of our approach, we studied 27 distributions of Firefox and analyzed one year of crash reports. The results of our analysis suggest that we can significantly reduce the number of submitted reports by bucketing 100 most frequent crash signatures at the client. This approach, besides enhancing users' privacy, improves the system utilization in terms of storage, network, and human resources by avoiding the transfer of redundant reports.

REFERENCES

- [1] 2010. Breakpad. Available at: <https://wiki.mozilla.org/Breakpad>.
- [2] 2010. Mozilla Crash Reporter. Available at: <https://support.mozilla.org/en-US/kb/mozillacrashreporter>.
- [3] 2010. Socorro. Available at: <https://wiki.mozilla.org/Socorro>.
- [4] 2010. Socorro: Mozilla's Crash Reporting System. Available at: <https://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports>.
- [5] 2011. Breakpad/Skip List. Available at: <https://wiki.mozilla.org/Breakpad/SkipList>.
- [6] 2012. Sampling Crash Volumes, Rates and Rarity for Socorro Samples. Available at: goo.gl/1v9PNZ.
- [7] 2016. Crash Signature Service. Available at: <https://github.com/adngdb/crash-signature-service>.
- [8] 2016. Debugging a Minidump. Available at: https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/Debugging_a_minidump.
- [9] 2017. A set of client and server components which implement a crash-reporting system. Available at: <https://chromium.googlesource.com/breakpad/breakpad/>.
- [10] 2017. About WER. Available at: [https://msdn.microsoft.com/en-us/library/windows/desktop/bb513613\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb513613(v=vs.85).aspx).
- [11] 2017. Analyzing Crash Reports. Available at: <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/AnalyzingCrashReports/AnalyzingCrashReports.html>.
- [12] 2017. Crash Reports. Available at: <https://www.chromium.org/developers/crash-reports>.
- [13] 2017. Decoding Crash Dumps. Available at: <https://www.chromium.org/developers/decoding-crash-dumps>.
- [14] 2017. Differential Privacy. Available at: https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.
- [15] 2017. Disabling Memory Dumps in the Event of a Crash. Available at: <https://www.safaribooksonline.com/library/view/secure-programming-cookbook/0596003943/ch01s09.html>.
- [16] 2017. Learn how identifiable you are on the Internet. Available at: <https://amiunique.org>.
- [17] 2017. Microsoft Symbol Server. Available at: <https://msdl.microsoft.com/download/symbols>.
- [18] 2017. Mozilla Crash Report. Available at: <https://crash-stats.mozilla.com>.
- [19] 2017. Mozilla Crash Reporters. Available at: <https://support.mozilla.org/en-US/kb/mozillacrashreporter>.
- [20] 2017. Mozilla Socorro Signature Utility Python Code. Available at: <https://bit.ly/386rBa7>.
- [21] 2017. Mozilla Symbol Server. Available at: <https://symbols.mozilla.org>.
- [22] 2017. Signatures Utilities Lists. Available at: <https://github.com/mozilla-services/socorro/tree/master/socorro/siglists>.
- [23] 2017. Super Search Documentation. Available at: <https://crash-stats.mozilla.com/documentation/supersearch>.
- [24] 2017. Understanding Crash Report. Available at: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Crash_reporting/Understanding_crash_reports.
- [25] 2017. User Identification. Available at: <https://wiki.mozilla.org/MetricsDataPing>.
- [26] 2018. About WER. Available at: <https://docs.microsoft.com/en-us/windows/desktop/wer/about-wer>.
- [27] Edward N Adams. 1984. Optimizing preventive service of software products. *IBM Journal of Research and Development* 28, 1 (1984), 2–14.
- [28] Iftekhar Ahmed, Nitin Mohan, and Carlos Jensen. 2014. The Impact of Automatic Crash Reports on Bug Triaging and Development in Mozilla. In *Proceedings of The International Symposium on Open Collaboration*. ACM, 1.
- [29] Amer Aljaedi, Dale Lindsog, Pavol Zavarsky, Ron Ruhl, and Fares Almari. 2011. Comparative analysis of volatile memory forensics: live response vs. memory imaging. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third International Conference on Social Computing (SocialCom)*, 2011 IEEE Third International Conference on. IEEE, 1253–1258.
- [30] Le An and Foutse Khomh. 2015. Challenges and issues of mining crash reports. In *Software Analytics (SWAN), 2015 IEEE 1st International Workshop on*. IEEE, 5–8.
- [31] Carina Andersson and Per Runeson. 2007. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering* 33, 5 (2007).
- [32] Kevin Bartz, Jack W Stokes, John C Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding Similar Failures Using Callstack Similarity. In *SysML*.
- [33] Pamela Bhattacharya and Iulian Neamtii. 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 1–10.
- [34] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. 2011. User tracking on the web via cross-browser fingerprinting. In *Nordic Conference on Secure IT Systems*. Springer, 31–46.
- [35] Justin Brickell, Donald E Porter, Vitaly Shmatikov, and Emmett Witchel. 2007. Privacy-preserving remote diagnostics. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 498–507.
- [36] Pete Broadwell, Matt Harren, and Naveen Sastry. 2003. Scrash: A system for generating secure crash information. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*. USENIX Association, 19–19.
- [37] Joshua Charles Campbell, Eddie Antonio Santos, and Abram Hindle. 2016. The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 269–280.
- [38] Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. [n.d.]. Automatically analyzing groups of crashes for finding correlations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM.
- [39] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. 2008. Better bug reporting with better privacy. In *ACM Sigplan Notices*, Vol. 43. ACM, 319–328.
- [40] 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 *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 1084–1093.
- [41] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 333–342.
- [42] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer* (2014).
- [43] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.
- [44] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. [n.d.]. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*.
- [45] Norman Fenton and James Bieman. 2014. *Software metrics: a rigorous and practical approach*. CRC Press.

- [46] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*.
- [47] 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.
- [48] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Dependable Systems & Networks (DSN), IEEE 41st International Conference on*.
- [49] Neda Ebrahimi Koopaei and Abdelwahab Hamou-Lhadj. 2015. CrashAutomata: an approach for the detection of duplicate crash reports based on generalizable automata. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 201–210.
- [50] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. Predicting the severity of a reported bug. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 1–10.
- [51] Carsten Maartmann-Moe, Steffen E Thorkildsen, and André Årnes. 2009. The persistence of memory: Forensic identification and extraction of cryptographic keys. *digital investigation* 6 (2009), S132–S140.
- [52] Tim Menzies, Ekrem Kocaguneli, Burak Turhan, Leandro Minku, and Fayola Peters. 2014. *Sharing data and models in software engineering*. Morgan Kaufmann.
- [53] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering*. IEEE.
- [54] Francisco Rocha and Miguel Correia. 2011. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Dependable Systems and Networks Workshops (DSN-W), 41st International Conference*. IEEE, 129–134.
- [55] Kiavash Satvat and Nitesh Saxena. 2018. Crashing Privacy: An Autopsy of a Web Browser's Leaked Crash Reports. *arXiv preprint arXiv:1808.01718* (2018).
- [56] Shaohua Wang, Foutse Khomh, and Ying Zou. 2013. Improving bug localization using correlations in crash reports. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 247–256.
- [57] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.
- [58] 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.
- [59] George OM Yee. 2011. *Privacy Protection Measures and Technologies in Business Organizations: Aspects and Standards: Aspects and Standards*. IGI Global.

A APPENDIX

Algorithm 1: CREPE-Client algorithm

```

1 receive (sigList)
2 receive (symList)
3 Crash Occur
4 try to build sig :
5 if debug.dmp == true then
6   build sig
7   if sig ∈ sigList then
8     submit report(crashid,sig,env)
9   else
10    submit fullreport(crashid,env,dmp,url,description)
11  end
12 else
13   submit fullreport(crashid,env,dmp,url,description)
14 end

```

Figure 11 demonstrates the distribution of frequent top crashes in Thunderbird in seven recent distributions. The result of our inspection shows a similarity pattern and statistically similar distribution between different versions (Figure 11).

Algorithm 2: CREPE-Server algorithm

```

1 Procedure: ServerProcess
2 developer.update(sigList)
3 developer.update(symList)
4 server.push(updates)
5 server.receive(report)
6 if report! = full then
7   add dmp to repository
8   update stats;
9 else
10  update stats;
11 end
12 Procedure: Server Pull Process
13 if developer.requestextradetails then
14   server.pull(report.dmp);

```

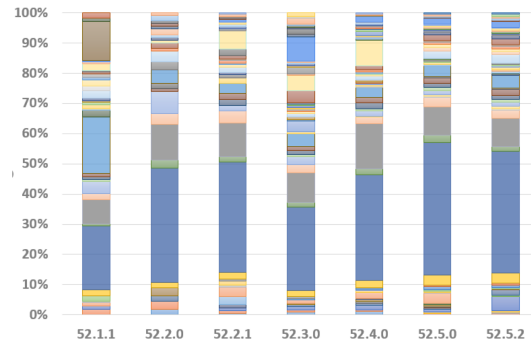


Figure 11: Distribution of top frequent signatures in Thunderbird across seven stable versions where each color represents a signature

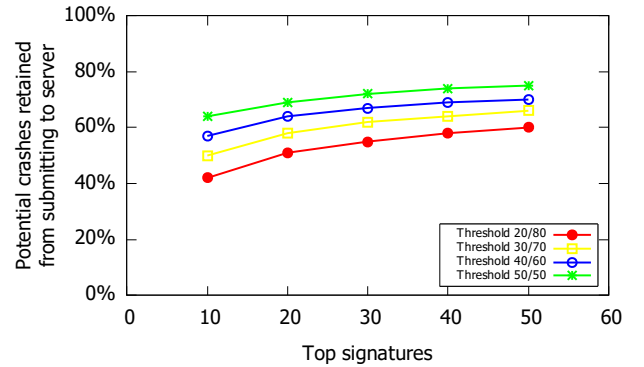


Figure 12: Potential reduction in the number of submitted crashes to the server based on the number of signatures that can be generated at the client in Thunderbird

A similar potential exists in the reduction of the number of submitted reports in Mozilla Thunderbird. However, the bug distribution, crash types, and the number of crashes occur in this application result in a different pattern in the distribution of signatures. As it can be seen from the Figure 12, a considerable portion of crashes belong to the top 10 crash signatures. Figure 12 illustrates the potential for reducing the number of redundant crashes in Thunderbird using different thresholds.