Crash Consistency Validation Made Easy

Yanyan Jiang*, Haicheng Chen†, Feng Qin†, Chang Xu*, Xiaoxing Ma*, Jian Lu*
*State Key Lab. for Novel Software Technology, Nanjing University, China
*Dept. of Computer Science and Technology, Nanjing University, China
†Dept. of Computer Science and Engineering, The Ohio State University, United States
jiangyy@outlook.com, {chen.4800,qin.34}@osu.edu, {changxu,xxm,lj}@nju.edu.cn

ABSTRACT

Software should behave correctly even in adverse conditions. Particularly, we study the problem of automated validation of crash consistency, i.e., file system data safety when systems crash. Existing work requires non-trivial manual efforts of specifying checking scripts and workloads, which is an obstacle for software developers. Therefore, we propose C^3, a novel approach that makes crash consistency validation as easy as pressing a single button. With a program and an input, C^3 automatically reports inconsistent crash sites. C^3 not only exempts developers from the need of writing crash site checking scripts (by an algorithm that computes editing distance between file system snapshots) but also reduces the reliance on dedicated workloads (by test amplification). We implemented C^3 as an open-source tool. With C^3, we found 14 bugs in open-source software that have severe consequences at crash and 11 of them were previously unknown to the developers, including in highly mature software (e.g., GNU zip and GNU coreutils sort) and popular ones being actively developed (e.g., Adobe Brackets and TEXstudio).

CCS Concepts

•**Software and its engineering** → **Software reliability;**

Keywords

File system, crash consistency, software reliability

1. INTRODUCTION

1.1 Crash Consistency Validation

Quality and reliable software is expected to behave correctly even in adverse conditions. Unfortunately, adverse conditions are relatively *infrequent* in practice and some may even be tricky that developers are not aware of their existence, leaving hidden "time bombs" in the software. Once such adverse conditions are triggered, the consequences can be entirely out of control–maybe as minor as a mobile-app crash (caused by an uncaught exception [31]) or as severe as causing billions dollars of economic cost (caused by a race condition in the blackout in 2003 [12]).

In this paper, we focus on the particular issue of *crash consistency* [26], which is an important property for any software that persists data. Crash consistency requires the application data (e.g., documents, data, and configurations) to be recoverable even if the system crashes [23]. Crash consistency is of significant importance because (1) as the software becomes widespread, any issue will eventually be exposed simply because of the Law of large numbers; and (2) crash inconsistency may lead to severe consequences. It would be shocking if your favorite document editor destroys your paper draft when your pet accidentally hits the hardware reset button at file saving.

However, developers oftentimes fail to provide crash consistency, as they often lack of knowledge on the crash behavior of the file operations and their underlying file system. Even experienced developers leave crash consistency bugs in their mature software systems [32].

Crash consistency can be validated using semi-automatic tools that simulate the crash behavior of a file system [22, 29, 32]. These approaches share a common workflow (Figure 1): (1) the program under test (e.g., a database implementation) is fed with test inputs or workloads; (2) the program execution's file system or I/O operations are logged; (3) simulated

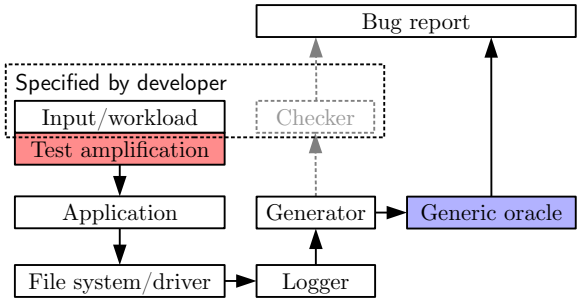

Figure 1: The workflow of crash consistency validation. White background cells denote the workflow of existing work [22, 29, 32]. In comparison, C^3 introduces generic oracle (blue), test amplification (red) and avoids user-specified checker (dashed grey).

[1]This work was done when Yanyan Jiang was a visiting student at The Ohio State University. Chang Xu and Xiaoxing Ma are the corresponding authors.

Table 1: Possible recovered crash sites of Ted text editor. A file is opened and saved once. Crash sites would be more complicated if multiple files are saved for multiple times and specifying of what are correct is a difficult task for its developer.

Category	*Original File Status*	*Backup File Status*	*Consistent?*	*Explanation*
C1	unmodified	N/A	✓	save operation failed with the undamaged original file
C2		corrupted		
C3	unmodified	up-to-date	✓	recoverable failure with the up-to-date backup file
C4	deleted			
C5	deleted	corrupted	×	data loss
C6	up-to-date	N/A	✓	the up-to-date original file

crash file system images (crash sites) are generated using the log; and (4) each crash site is checked against a manually specified checking script (e.g., the database's ACID checker) for crash consistency.

While being effective in disclosing crash consistency bugs in data storage systems with well-defined crash semantics, existing approaches heavily rely on developers' manual efforts, making them cumbersome to use in practice. To make crash consistency validation easy, the first challenge (test oracle) is that existing techniques require developers to specify checking scripts to determine consistency of a crash site. However, they have no idea of crash consistency and are not trained for specifying such property. The second challenge (test input) is that the existing test inputs, usually test cases for functional validation, may not be sufficient to reveal crash consistency bugs.

1.2 How to Make it Easy?

In this paper, we propose the Crash Consistency Checker (C^3 for short) to *automatically* validate crash consistency of application software. C^3 makes crash consistency validation as easy as clicking one button. Given a program with an input (usually a simple use case), C^3 either certifies the execution to be crash-consistent or reports an inconsistent crash site for further inspection. We present the motivation, the challenges, and the overview of C^3 in Section 2.

To *remove the manual efforts of writing checking script*s, we devise a *generic test oracle* at file system level (instead of application level) for validating whether a crash site is consistent. We observed that developers usually expect the file system snapshot after a meta-data operation (e.g., directory operations, file close, and `fsync`) to be consistent. Accordingly, we define crash sites that can be aligned with such a consistent snapshot via simple recovery operations to be consistent. The oracle fully automates the crash consistency validation procedure, exempting developers from writing the checking scripts that relates to the program semantics.

Automating the validation is not sufficient to reveal many crash consistency bugs. We observed that a file system implementation may enforce crash consistency by chance (e.g., truncate and overwrite a small file), causing both manual-specified checking script and our generic oracle to miss a crash consistency bug. Such bugs can only be manifested by dedicated workloads (e.g., a sufficiently large input file), which are difficult for developers to provide in practice.

To *reduce the reliance on dedicated workloads*, we further propose *test amplification* that injects benign file system synchronization operations in the middle of a program execution to break such accidental atomicity so that our generic test oracle can disclose more crash consistency bugs. Technical details are discussed in Section 3.

We implemented C^3 prototype tool, made it publicly available and open-source and hope it will help developers discover more crash consistency bugs early. Our C^3 implementation adopts non-intrusive system call instrumentation and virtual device that are transparent to the program and the file system. Almost any program written in any language running on any file system can be validated by our tool. The implementation decisions are discussed in Section 4.

We conducted experiments on 25 popular open-source programs to evaluate the effectiveness and efficiency of C^3. We discovered crash consistency bugs in 14 subjects, where 11 were previously unknown (7 cannot be manifested without our test amplification). All the bugs lead to severe consequences (data loss or corruption). Some bugs are from highly mature software (e.g., GNU zip, GNU coreutils sort) or from popular ones under active development (e.g., GitHub Atom, Adobe Brackets, and jEdit). Evaluation results also show that C^3 is easy to use and consumes affordable resources. The evaluation is presented in Section 5.

Finally, we summarize the lessons learned in our experiments and communications with the open-source community in Section 6, followed by related work and conclusion in Sections 7 and 8, respectively.

2. C^3 IN A NUTSHELL

2.1 Two Motivating Examples

We demonstrate the challenges of crash consistency validation by two real-world crash consistency bugs discovered by C^3, both were previously unknown and have severe consequences. Inconsistent crash sites can be manifested on a typical Linux distribution with ext4 file system with default settings, which represents a typical user's environment.

2.1.1 Ted Text Editor

Productivity software manages user data such as documents, photos and settings. Such contents should be handled with extreme care, as corrupting them can lead to catastrophic consequences. The following (simplified) file-saving code is from the Ted text editor on Android, which has more than 100K installations:

```
1 backupPath = path + ".tmp";
2 TextFileUtils.writeTextFile(backupPath,
      content);
3 deleteItem(path);
4 renameItem(backupPath, path);
```

At a first glance, this code seems to be crash-safe: file contents are first saved to a temporary file; the original file is then deleted, followed by a renaming of the temporary file. The developer expected that, whenever the system crashes, at least one from the original file and the backup file will remain in the file system.

Such expectation lays on the assumption that each file system operation is processed *in-order* and immediately takes effect. Unfortunately, this assumption is not valid as both file system and device driver reorder requests for performance. There is no ordering guarantee between the system calls `write` (in `TextFileUtils.writeTextFile`) and `unlink` (in `deleteItem`), causing the original file being deleted before the backup file contents are persisted. The file system after crash recovery may contain a single corrupted file (0 byte), indicating a catastrophic data loss.

To ensure the ordering between such file system operations in the example, one can either insert a file system flush call (`fsync`) after the file contents being written or remove the `deleteItem` line (ext4 file system on Android provides strong consistency guarantee for renaming). Our patch is already merged by the developers[1].

This example demonstrates the challenge of automatically deciding consistency of a crash site. In the example, even the simplest use case has six categories of crash sites (Table 1) and only one of them is inconsistent. Validating crash consistency is a non-trivial task for software developers because (1) developers often do not master the knowledge of crash consistency; (2) there can be many accesses to multiple files and consistency should be defined for every potential intermediate state; and, (3) file system accesses can be in libraries that are not well-understood. We address the challenge by proposing a generic oracle for automated crash site validation.

2.1.2 GNU coreutils sort

Sort is a command-line tool that prints input lines in sorted order, which appeared in the first version of Unix and is now provided by GNU's core utilities.

Sort is mostly used by pipeline and redirection. However, it still provides an option to write the results to a destination file (for backward compatibility of the old-time sort that sorts a file in-place). The developer community also considers sorting files in-place using `sort data -o data` a valid option and this solution received the highest votes by the viewers on StackOverflow[2].

Unfortunately, the practice (or option) of overwriting the source file for in-place sorting leads to potential data loss when the system crashes. The destination file is first opened and truncated to empty. If at this time system crashes or the disk runs out of space, contents in `data` are permanently lost. We reported this issue to the developers. They indicated that a completely safe and portable solution is difficult to work out (due to permission, owner, and hard-link issues) and the bug is currently fixed by explicitly documenting this dangerous behavior[3].

Surprisingly, if the input file is of small size, this bug cannot be triggered on the ext4 file system with the default setting. In all possible crash sites of a profiling run, the data file is either unmodified or up-to-date, because the file system implementation ensures the atomicity of a small-file overwrite. Neither manually-specified checkers nor our generic oracle can detect the crash consistency bug, unless dedicated workloads (e.g., large files) are provided.

This example demonstrates the challenge of reducing the reliance on the dedicated workloads. Among all existing approaches [22, 29, 32], only Alice [22] has a chance to detect the bug without dedicated workloads. However, it relies on the correct abstract model of file systems. Alternatively, we address this challenge by test amplification that injects benign events in the middle of program execution.

[1]https://github.com/xgouchet/Ted/pull/45.
[2]http://stackoverflow.com/questions/9117274.
[3]http://debbugs.gnu.org/cgi/bugreport.cgi?bug=22769.

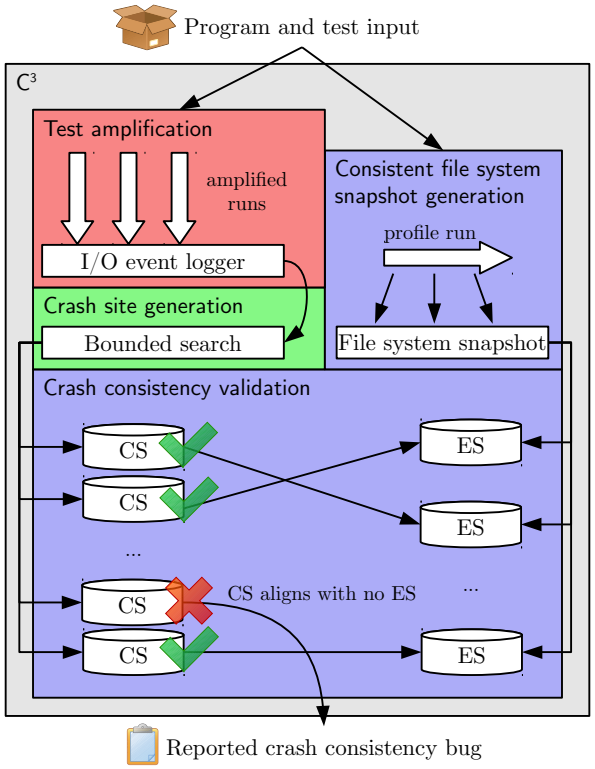

Figure 2: Architectural overview of our C³ approach. Blue, red and green components are discussed in Sections 3.1–3.3, 3.4 and 3.2, respectively.

2.2 Workflow of C³

C³ adopts a methodology in crash consistency validation similar to that of storage stacks [22, 29, 32], decomposing the problem into three sub-problems:

1. (Test input) Find suitable inputs/workloads that can reveal potential crash-inconsistent vulnerabilities.
2. (Crash site generation) Derive possible crash sites for each test run.
3. (Test oracle) Validate each crash site's consistency.

We explain the workflow of C³ in the following. The architectural overview of C³ is shown in Figure 2. The technical details are expanded in Section 3.

2.2.1 Test Input

The validation procedure of C³ is driven by the software's functional tests/use cases which are easy to obtain. Recall that existing work use dedicated workloads to find bugs

in storage stacks [22, 29, 32]. Such workloads are usually overkill for validating application software because applications have much simpler file system access patterns than a storage system (e.g., a database or a version-control system). C^3 also adopts test amplification to further reduce the reliance on dedicated workloads because file system implementation may keep atomicity of operations by chance (Section 2.1.2).

2.2.2 Crash Site Generation

C^3 takes the standard approach [29, 33] to generate simulated crash sites by intercepting I/O requests at runtime using a virtual RAM disk. We formally define the semantics of I/O requests and use this definition to derive all possible crash sites based on the specification of Linux block layer. Based on the observation that inconsistency can mostly be manifested by dropping a small number of metadata blocks, C^3 uses a bounded-search algorithm to generate crash sites. The algorithm enumerates the crash points and systematically drops a subset of the blocks, yielding crash sites to be validated. Each crash site is mounted in the local file system and validated by our generic oracle.

2.2.3 Test Oracle

The key idea of C^3 generic oracle is to make the expectation of software developers explicit, defining a set of consistent file system snapshots. Our observation is that programs are usually crash-consistent if file system operations has *atomicity* and *persistence* (otherwise the bug can be manifested without system crash, which is out of our scope). Accordingly, C^3 collects file system snapshots after metadata operations (directory operations, file close and `fsync`) and consider them to be consistent.

C^3 certifies a crash site to be consistent if it has a small editing distance to a consistent snapshot, i.e., it can be transformed to a consistent snapshot via a series of simple recovery operations that do not involve out-of-thin-air content creation. In other words, starting from such a crash site, software users can easily fall back to a consistent state. Realizing that exact computation of editing distance is intractable, we adopt an alternative relaxed necessary condition that can be efficiently computed in C^3.

3. VALIDATING CRASH CONSISTENCY

In this section, we provide in-depth discussion of C^3 in a slightly different order than its workflow. We first discuss the definition of expected file system snapshots (ESs) and why they are consistent in Section 3.1, followed by how to obtain crash file system snapshots (CSs) in Section 3.2. Then, we show how to validate the consistency of a CS in Section 3.3. Finally, we discuss the design of test amplification in Section 3.4.

3.1 Defining Consistent File System Snapshots

When developers are manipulating user-generated contents, they usually do have considerations of data safety (e.g., Ted intentionally writes to the backup file). Therefore, the gap between developer's expectation and actual implementation of file system consistency can lead to crash consistency bugs. File systems only provide simple interface for data management and do not have transactional semantics. As a result, the highest level of file system consistency is *atomicity* and *persistence* of system calls, as if each is issued in-order and immediately persisted to disk. File system implementations do provide such consistency semantics to its users, assuming the system never crashes. However, such consistency breaks at system crash because both file system and device driver are allowed to buffer and reorder I/O requests for maximized performance [22]. Unfortunately, this phenomenon is not well-understood by the developers and becomes the root cause of (unrealistic) expectations. For example, Ted developers expected buffered data to be always persisted to disk before file deletion takes effect but this is unfortunately the case in the target platform's file system implementation (ext4).

Following this intuition, we assume that the program correctly handles crashes on a strongly consistent file system. In other words, we assume that developers do not make obvious data-safety mistakes that can be triggered without any system crash, e.g., deleting original file before backup is written[4]. Particularly, it is reasonable to believe that developers have knowledge of the intermediate state after every *meta-data* operation (directory operations, file close and `fsync`) because these operations cause significant changes to the file system state.

Therefore, we consider file system snapshots of such intermediate states to be "expected" reference snapshots (Expected Snapshots, or ESs). ESs serve as the basis of defining consistency of a crash site. ESs are collected by a profiling run in which system calls are intercepted. After each file metadata operation, we pause the program and traverse the file system to obtain its snapshot. Back to the Ted example (Table 1), each of Categories 1, 3, 4, and 6 corresponds to a consistent file system snapshot after meta-data operation: program start (C1), close of backup file (C3), deletion of original file (C4), and renaming (C6).

An ES consists of files in directories. We define an ES to be a set of tuples $\{\langle f_1, c_1\rangle, \ldots \langle f_n, c_n\rangle\}$, where f_i denotes i-th file's full path (e.g., `/mnt/crashdisk/file.txt`) and c_i denotes its contents. We flatten the tree structure because crash consistency focuses on safety of file contents. A file f_i with contents $c_i = [b_1, b_2, \ldots, b_m]$ denotes that its size is m bytes and j-th byte value is b_j. The profiling run returns $\mathcal{E}$, the set of all ESs.

3.2 Generating Crash Sites

To generate crash sites for validation, we do not actually power off the machine. Rather, we take the standard approach of existing work [22, 29, 32] by keeping a log of I/O requests performed by the program and synthesizing crash sites at simulated crash points.

Physical disks are *not* required to process I/O requests in their arrival order for performance, which is a major source of crash inconsistency [7, 22]. When file system requires ordering between operations, it invokes a *disk barrier* (indicated by a `REQ_FLUSH` or `REQ_FUA` flag in a Linux block I/O request) to flush pending requests to disk.

To capture the effect of all possible request ordering, disk semantics is formalized as follows. A disk D is a mapping from sector identifier to its actual stored data. For each sector $s \in \{1, 2, \ldots\}$, we use $D(s)$ to reference its data. At runtime, there is an internal queue Q of requests pending to be flushed as well as auxiliary mappings U and V. $U(i)$ and $V(i)$ denotes i-th request's sector identifier and data,

[4]There can be data loss if the program is killed in the middle.

Algorithm 1: Crash snapshot generation algorithm

Input: A sequence of I/O requests $\{e_1, e_2, \ldots, e_n\}$ and a search bound k

Output: A set $\mathcal{C}$ containing crash snapshots

```
1  C ← ∅;
2  for j ∈ {1,...,n} ∧ e_j is not a barrier do
3      Let (D, Q = {e_i, e_{i+1}, ... e_j}, U, V) be the state after
       executing events {e_1, e_2, ..., e_j}
4      for ℓ ∈ {0, 1, ..., min{k, j − i + 1}} do
5          for P ⊆ {0, 1, ..., j − i − 1} ∧ |P| = j − i − ℓ do
6              D_c ← D;
7              for s ∈ P ∪ {j − i} do
8                  D_c ← D_c[U(i + s) ↦ V(i + s)];
9              if D_c ∉ C then
10                 S ← mount(D_c);
11                 C ← C ∪ {S};
```

respectively, and both are initially empty. A sequence of *write* and *barrier* requests[5] $\{e_1, e_2, \ldots, e_n\}$ are allowed to be performed on a disk with the following semantics.

1. A write request $\mathsf{W}(s, d)$ to s-th sector with data d. Rather than being immediately persisted, the request is queued in Q. The notation $A[x \mapsto y]$ denotes map replacing, i.e., $A[x \mapsto y] = A \setminus (x, A(x)) \cup (x, y)$:

$$\frac{e_i = \mathsf{W}(s, d)}{(D, Q, U, V) \Rightarrow (D, Q \cup \{e_i\}, U[i \mapsto s], V[i \mapsto d])}.$$

2. A barrier request B ensures all write operations before it to be persisted:

$$\frac{\begin{array}{c} e_i = \mathsf{B} \\ Q = \{e_p, e_{p+1}, \ldots e_q\} \\ D_p = D \\ D_{k+1} = D_k[U(k) \mapsto V(k)]\ (k \in \{p, p+1, \ldots, q\}) \end{array}}{(D, Q, U, V) \Rightarrow (D_{q+1}, \varnothing, U, V)}.$$

Finally, at any system state (D, Q, U, V), we allow the system to crash, yielding a set of valid crash disks $\mathcal{D}_{\text{crash}}$:

$$\frac{\begin{array}{c} Q = \{e_i, e_{i+1}, \ldots e_j\} \\ \mathcal{D}_i = \{D\} \\ \mathcal{D}_{k+1} = \mathcal{D}_k \cup \{D[U(k) \mapsto V(k)] \mid D \in \mathcal{D}_k\} \end{array}}{(D, Q, U, V) \Rightarrow \mathcal{D}_{\text{crash}} = \mathcal{D}_{j+1}}.$$

This crash model describes the exact contract between a disk driver and the Linux block I/O layer. It defines all possible outcomes of reordering (from a disk's perspective, any effect of reordering is equivalent to dropping a subset of requests). Furthermore, our crash model assumes the physical disk to be *reliable*, i.e., persisted data never corrupts. Otherwise, unreliable physical disk (e.g., severe faults studied in [33]) may lead to crash sites that cannot be recovered.

Algorithm 1 displays the crash site generation algorithm. Exhaustively enumerating all crash sites [29] is too time-consuming. Instead, we use a bounded-search algorithm based on the observation [32] that dropping only a few of critical I/O requests can manifest crash consistency bugs. We accordingly enumerate the point of system crash (Line 2) and those crash sites who drop at most k requests in the pending queue Q (Lines 4–8). The search bound k is adjustable: if the time budget is limited, we can bound k to be a small constant. A sufficiently large k is equivalent to an exhaustive enumeration.

Generated disk images are mounted to the native file system and further checked for crash consistency. A crash file system snapshot (Crash Snapshot, or CS) is similar to an ES described in Section 3.1: the crash snapshot $S = \{\langle f_i, c_i\rangle\}$ denotes that file f_i has a contents of c_i.

3.3 Validating Crash Sites

The key insight of our general oracle is based on the following case analysis of a crash site S:

1. S is identical to a consistent ES $S' \in \mathcal{E}$. Our basic assumption of ES implies that S is consistent.
2. We can transform S to $S' \in \mathcal{E}$ by performing simple recovery steps that do not involve out-of-thin-air content creation. An example is C2 of Table 1 where a corrupted file may contain partial data and deleting corrupted backup file yields a consistent state. If we can obtain a consistent state from S regardless of program semantics, S should be consistent.
3. Neither (1) nor (2) applies. In this case, non-trivial recovery scheme is required to fall back to a consistent state. If such scheme does not exist for general application software, S is highly likely to be inconsistent.

This trichotomy yields a definition of crash consistency based on the *editing distance* [21] that avoids both false positives (reporting a recoverable non-ES crash site as inconsistent) and false negatives (failing to report inconsistent CSs). Formally, for a crash site $S = \{\langle f_i, c_i\rangle\}$ to be consistent, there must exists an $S' = \{\langle f'_i, c'_i\rangle\} \in \mathcal{E}$ such that S can be transformed to S' using a bounded number of following editing operations (assume that $\langle f, c\rangle \in S$, $c = [b_1, b_2, \ldots, b_m]$ and f' can be arbitrary file-name other than f):

1. Creation of an empty file: $S \Rightarrow S \cup \langle f', \varnothing\rangle$.
2. Deletion of a file: $S \Rightarrow S \setminus \langle f, c\rangle$.
3. Renaming of a file: $S \Rightarrow S \setminus \langle f, c\rangle \cup \langle f', c\rangle$.
4. Moving a consecutive segment of file contents: $S \Rightarrow S \setminus \langle f, c\rangle \setminus \langle f', c'\rangle \cup \langle f, [b_1, \ldots, b_m, b'_p, \ldots, b'_q]\rangle \cup \langle f', [b'_1, \ldots, b'_{p-1}, b'_{q+1}, \ldots b'_{m'}]\rangle$, where $[b'_p, \ldots, b'_q]$ is a substring in the contents of file f'.

This definition echoes the trichotomy: a CS is consistent only if it can be aligned with an ES with a small editing distance. Otherwise, a large or infinite[6] editing distance indicates an impossible or highly non-trivial recovery and we have sufficient evidence to report it as inconsistent. Unfortunately, this particular version of editing distance is intractable. Interested readers can refer to our NP-Completeness proof in Appendix A.

We discovered that a *relaxed* definition of alignment is already sufficient for crash consistency validation and can be efficiently computed. Particularly, we define a CS S to be consistent if S can be transformed to an ES S' with a *finite* number of editing operations.

This relaxed definition is equivalent to the existence of an injective mapping from every byte in the files of S to a byte in that of S', simply because an unbounded number of

[5]Reads do not affect contents in the disk and we do not consider them in defining crash semantics.

[6]If it is impossible to transform a CS to an ES, the editing distance is infinite.

editing operations allows bytes in S to be arbitrarily permuted, redistributed, and deleted. This relaxed property is also much easier to check. Formally, S is consistent only if there exists $S' \in \mathcal{E}$ such that for every byte value σ,

$$\sum_{\langle f,c\rangle \in S'} \Big|\{j \mid c_j = \sigma\}\Big| \leq \sum_{\langle f,c\rangle \in S} \Big|\{j \mid c_j = \sigma\}\Big|.$$

This alternative definition of alignment naturally gives a linear time validation algorithm by comparing the number of each byte value's occurrences.

Finally, we argue that the relaxation is also effective in crash consistency validation. First, whenever a CS cannot be aligned with an ES in the relaxed definition, the editing distance must be infinite. Therefore, as long as the editing distance reports no false positive of crash inconsistency, so does the relaxed definition. In theory, the relaxed definition may misclassify an actually inconsistent CS as consistent, leading to potential false negatives. However, this is expected to be rare in practice, as reporting crash inconsistency only requires *one* witness and the relaxed condition fails to detect the issue only if it reports false negative on *all* CSs. There likely exists at least one inconsistent CS that is largely corrupted (e.g., file contests are mostly corrupted), so our relaxed condition tends to capture it and report the crash consistency bug.

3.4 Amplifying Test Inputs

In the GNU coreutils sort example (Section 2.1.2), the crash consistency bug cannot be manifested without dedicated workloads because the file system implementation ensures small-file overwrite's atomicity by chance. However, such atomicity is not a guaranteed offer. If the file is sufficiently large, we can observe inconsistent crash sites that only contain partial data and cannot be aligned to any ES.

To exempt the need of dedicated workloads (e.g., huge input files), we designed a *test amplification* approach. Recall the root cause of the hidden bug is (not guaranteed) atomicity of consecutive operations, we break such atomicity by injecting system-wide synchronization operation (`sync`) in the middle of a program execution. Such operations are totally benign, i.e., do not affect the application view of the file system, but can manifest the inconsistent intermediate crash sites.

Test amplification is conducted in a single separated program execution called *amplification run*. In the amplification run, we intercept system calls that may silently lead to data loss (`ftruncate` and `open`, which are usually contained in library code of which developers are not aware) and inject a `sync` after each of them. The I/O request log collected for the amplification run is used for further crash site generation (Section 3.2) and consistency validation (Section 3.3).

In the GNU coreutils sort example, test amplification injects a `sync` after the data file is truncated, yielding a CS that contains only an empty data file, which cannot be aligned to any ES and is correctly reported as a crash consistency bug.

4. IMPLEMENTATION

We implemented our C^3 approach as a prototype tool and made it public and open-source[7]. Both the instrumentation and the I/O requests logger in C^3 are transparent to the file system. Therefore, C^3 can validate software written in any language, using any libraries, and running on any file system. The idea of C^3 can be also implemented on other systems (e.g., by using a simulated iSCSI device [32]). The rest of this section expands discussion of techniques used in our C^3 implementation.

[7]Available at http://jiangyy.github.io/c3/.

```
1  @prepare
2  def init_setup():
3      prepare_init_file()
4  @run_program
5  def start_program():
6      os.system("brackets") # execute program
7  @delay(5.0)
8  def do_edit():
9      edit_document()
10     keypress('ctrl-s') # save document
11     keypress('alt-f4') # exit program
```

Figure 3: Simplified test script for the `Brackets` text editor in which we found crash consistency bug.

4.1 Test Input

C^3 runs the program multiple times using the same test inputs. Test inputs are specified by test scripts, which are based on a series of decorated functions in Python (Figure 3). A test script provides means to specify (1) an initial file system snapshot; (2) how to load the program; and, (3) actions to be performed at program runtime. To further ease the testing procedure, we also developed a simple record tool [16] that captures system-level UI events and automatically synthesizes a test script for GUI software.

For each amplification run, C^3 instruments the program using `ptrace`, intercepting the program's control flow whenever a system call is about to execute. C^3 injects a synchronous `sync` call if a designated point is reached and then resumes the program execution.

In this paper, we do not focus on how such inputs are obtained. Even though the program may be large and complicated, there usually are only a few places that interact with the file system. We believe that simple use cases are sufficient to reveal many crash consistency bugs and developers will have no obstacle providing test inputs that cover all file system operations.

4.2 Crash Site Generation

C^3 collects a I/O request log for crash site generation by a virtual RAM disk driver, which is similar to `eXplode` [29]. Before executing the test script, the virtual disk issues an `ioctl` call to the driver for capturing the initial snapshot of the virtual disk. During the test script execution, the virtual disk handles I/O requests like a normal RAM disk and at the same time keeps an internal copy of all write and barrier requests. After the termination of the test script, these logged data are dumped back to user space via another `ioctl` call and CSs are generated using Algorithm 1.

C^3 can only validate consistency of file system snapshots on the virtual RAM disk. Therefore, test scripts should place files to be manipulated on the virtual disk. However, the program may also modify files whose paths are hard-coded to the local file system (e.g., `/install/path/.config`).

If developers also intend to validate crash consistency of such files, they can create its shadow copy on the virtual disk and replace the file in the local file system by a symbolic link.

4.3 Test Oracle

C³ collects a file system snapshot after each file system metadata operation and considers such ESs to be consistent. ESs are collected at a separate profiling run in which system calls are instrumented by `ptrace`. ES cannot be obtained by the virtual disk (file system calls do not take effect immediately). Rather, the file system itself always has its consistent "current" view and C³ peeks the file system by a simple read-only traversal of the file system. Furthermore, C³ does not consider a file system snapshot that contains no data to be consistent, such that we can detect bugs caused by developer's accidental deletion of a file. Finally, the atomicity of an ES collection is guaranteed via serializing file system calls by `ptrace` and restricting the program to be the only process that can access the virtual disk.

Each generated CS is mounted for crash consistency validation. A CS is scanned to obtain each byte value's statistics and these values are compared against those of ESs to decide whether the CS is consistent. Though the time complexity of consistency checking is linear, we still adopt fingerprinting to further improve the efficiency of C³. We associate each disk image (both CSs and ESs) with the hash fingerprint[8] of its contents. ESs are de-duplicated according to their fingerprints and the CSs that have been validated are immediately skipped to reduce time cost.

4.4 Putting Them Together

C³ combines all techniques discussed in this section to realize the workflow in Figure 2 and creates crash consistency bug reports. C³ runs the program three times: a profiling run to collect ESs and two test runs (a normal run and an amplification run) to collect CSs for crash consistency validation. For each inconsistent CS, we calculate d to be the *minimum* editing distance to an ES (in our relaxed definition, d denotes the number of bytes that cannot be aligned). C³ reports the crash site that has a *maximum* d as the inconsistent crash site for the program.

The reported crash site has the most bytes that cannot be aligned and thus is most likely to be inconsistent. If it is indeed inconsistent (i.e., a true positive), developer can fix the problem and run C³ again for further validation. Otherwise, the CS cannot be transformed to an ES using simple recovery operations, suggesting that a checking script is needed.

Furthermore, we only report an inconsistent CS when $d \geq 32$ to reduce false positives caused by a small degree of non-determinism. An example is a program that writes the timestamp into file's contents. In this case, any CS in the test run would not align with an ES in the profiling run (i.e., $d = 0$) because timestamp in the CS and the ES are distinct. If a CS is inconsistent, it has at least one sector of data to be corrupted, which likely to yield more than 32 bytes of data (6.25% of a sector with the size of 512 bytes) that cannot be aligned. Therefore, this treatment both reduces false positives and has negligible probability of missing a truly inconsistent CS.

[8]We keep 160-bit SHA-1 fingerprints such that the probability of hash collision is negligible.

5. EVALUATION

5.1 Methodology

We evaluated the effectiveness, ease-of-use, and performance of C³ using real-world software and typical use cases. For effectiveness (Section 5.2.1), we study (1) whether C³ can discover crash consistency bugs in real-world software, which is demonstrated by an empirical study of bugs found; (2) whether test amplification is effective in detecting crash consistency bugs, which is denoted by the percentage of bugs that require test amplification to manifest; and, (3) whether C³ reports false positives, which is presented by a qualitative study. We also evaluated more subjects studied by Alice [22] to compare the effectiveness of the C³ oracle with the manual checking scripts.

For ease-of-use (Section 5.2.2), we study whether the manual efforts to use C³ are minor. C³ only asks the developer to specify a use case and the subsequent test amplification and crash consistency validation is fully automated.

For performance (Section 5.2.3), we study whether the cost of C³ is practically affordable. We measure the following quantities for each run of C³: number of ES collected, number of CS checked, and time consumed in each phase.

We evaluate C³ using 25 applications from two categories: *utilities* for command-line use (e.g., make, gzip and indent, 10 in total) and *productivity* applications for editing user generated contents (e.g., TEXstudio, Atom and Libreoffice, 15 in total). We select these applications because they manipulate file system data and are of significant popularity based on authors' experiences and Internet search results.

For each subject, we specify *one* typical use case (either from the documents or from authors' daily use) in the test script format (Section 4.1). The use case represents the most common usage of the software in which we believe the crash consistency bug will have the most severe consequences. As C³ is publicly available, developers can easily validate crash consistency for any corner case test input.

For each use case, we validate its crash consistency by C³. If C³ reports an inconsistent CS (C³ always outputs the crash site with the most bytes that cannot be aligned), it is manually analyzed by studying the source code and the system call trace. True positives (i.e., crash consistency bugs) are reported to the developers.

All evaluations were conducted on a commodity environment for software users: a virtual machine with two virtualized Intel i5 CPUs and 2GB of RAM running Ubuntu Linux 14.04 (Kernel 4.2). The virtual disk is formatted with ext4 of default options, which denotes the most prevalent file system setting. Developers can also validate crash consistency under other file system settings by modifying only one line in the C³ configuration.

5.2 Evaluation Results

5.2.1 Effectiveness

Bugs found. We summarize bugs found by C³ in Table 2. Among 25 evaluated subjects, even if each subject is evaluated by one simple use case, C³ reported 14 inconsistent crash sites (all have a sufficiently large editing distance d exceeding the threshold). All crash sites are manually analyzed with the system call traces in which we confirmed the existence of data loss or corruption. Therefore, we submitted the bug reports for all 14 subjects and the developers

Table 2: Crash consisntecy bugs discovered by C³. Bug# denotes the bug/issue ID in the issue tracking system. A bold bug# indicates a previously unknown bug. Amp. column indicates the bug can only be manifested with test amplification.

Type	*Application*	*LOC*	*Language*	*Version*	*Bug# (tracker)*	*Amp.*	*Consequence*	*d (bytes)*
Utility	GNU make	39.0K	C	4.1	46193 (savannah)		Incorrect build	7.33K
	GNU zip	47.3K	C	1.6	**22770** (debbugs)		Data loss	5.04K
	bzip2	8.12K	C	1.0.6	**N/A** (email)		Data loss	8.56K
	GNU coreutils sort	4.65K	C	8.21	**22769** (debbugs)	✓	Data loss	23.9K
	Perl	801K	C	5.22	**127663** (perlbug)		Data loss	17.4K
	Shelve	0.23K	Python	2.7.11	**25442** (bug tracker)		Corruption	907
Productivity	Gimp	522K	C	2.8.14	763124 (bugzilla)	✓	Data loss	188K
	CuteMarkEd	21.8K	C++	0.11.2	**285** (github)	✓	Data loss	5.61K
	TEXmaker	46.7K	C++	4.5	**1553361** (launchpad)	✓	Data loss	1.61K
	TEXstudio	139.6K	C++	2.10.8	**1693** (sourceforge)	✓	Data loss	1.61K
	Ted	3.7K	Java	1.0	**45** (github)		Data loss	4.10K
	jEdit	188K	Java	5.1.0	**3952** (sourceforge)		Data loss	1.61K
	GitHub Atom	55.8K	Node.js	1.5.3	10609 (github)	✓	Data loss	1.61K
	Adobe Brackets	117K	Node.js	1.5.0	**12103** (github)	✓	Data loss	1.61K

confirmed 8 as previously unknown bugs and 3 as previously known bugs (e.g., the bug is fixed in the current development branch but our validation is based on the latest stable release). The remaining 3 bug reports have yet received any responses, but we believe they were also previously unknown based on the search results in the bug/issue tracking system.

All bugs found by C³ have severe consequences like data loss or corruption, which are analyzed as follows. In 12 out of 14 bugs (gzip, bzip2, sort, perl and all productivity subjects), the user's file or data can be completely lost after crash. Furthermore, such bugs were triggered in practice. For example, GitHub Atom users manifested the same bug in another adverse condition: when the disk runs out of space at the halfway of file saving. Gzip developers also believe data loss had happened before, however, the bug is not reported maybe due to its irreproducibility.

For the Python standard library Shelve, the bug leads to corrupted database that cannot be analyzed. The library provides three backends for data storage but C³ found that none of them is crash-safe. One of such backends is GDBM whose crash consistency bug is also discussed in [22]. Some developers believe that a SQLite backend should be provided for data safety.

For GNU make, we validated the use case of incremental build of `foo.c`. The inconsistent crash site contains a corrupted `foo.o` whose timestamp is up-to-date. If we proceed with incremental build after system crash, `foo.c` will be ignored. Such behavior leads to failed (e.g., fail to link a corrupted build target) or erroneous build (e.g., corrupted file packed into the package). Implications of these real-world bugs are further studied in Section 6.

Finally, bugs reported by C³ also received positive feedbacks from the open-source community. After we reported the bug of gzip in the mailing list, the developers of lzip (a functional equivalent of gzip) confirmed that lzip has the same crash consistency bug.

These results evidently support that C³ is effective and promising in crash consistency validation.

Test amplification. Column 7 of Table 2 shows that half (7/14) of the bugs cannot be manifested without test amplification using simple test inputs.

An interesting case is TEXstudio, which (1) writes the file contents to a temporary file; (2) opens the original file with `O_TRUNC`; (3) unlinks the temporary file; and, (4) writes the file contents to the original file. Surprisingly, the file system implementation postpones the effect of unlink and this seemingly-obvious data loss bug cannot be observed in any possible crash site unless the file is huge. With C³'s test amplification, we can discover the inconsistent crash site that only contains a truncated file.

These results indicate that the test amplification is effective in exempting the need of dedicated workloads.

False positives. We did not observe any false positive in the evaluated subjects. However, C³ may report false positives if the crash recovery requires non-trivial efforts, e.g., in validating databases [32].

Nevertheless, false positive may not be a big issue for software developers because C³ reports inconsistent CS with an explanation (the CS cannot be easily transformed to an ES). By examining the CS and ES, the developer can quickly pinpoint the root cause of false positives and provide additional rules to filter out actually consistent crash sites that is reported by the generic oracle of C³.

Comparisons with manual checking scripts. We evaluated more subjects studied in Alice [22]. These crash consistency bugs are discovered by manual checking scripts. We ran them with C³ using simple workloads. For GDBM [1], C³ correctly reported a corrupted database file. For LevelDB [3] and LMDB [4], C³ reported false positives–inconsistent snapshots that have a relatively small $d \leq 256$. For the reported crash sites, running the default database recovery will obtain a consistent database. For SQLite [6], C³ considers it as consistent and missed the durability bug because it relates to the database's semantics. For Git [2] and Mercurial [5], C³ did not found crash inconsistency. The system call trace study [22] suggests that crash may lead to corrupted data (cannot be opened), but data is not actually lost and may be recovered by an experienced user. These results are expected because C³ trades, to some degree, the effectiveness of the tool (detecting more bugs by learning the semantics of each application) for the easy-of-use of the tool (automating the crash consistency checking procedure).

5.2.2 Ease of Use

We demonstrate that the efforts of using C³ are minor and trivial for developers. To validate crash consistency, a devel-

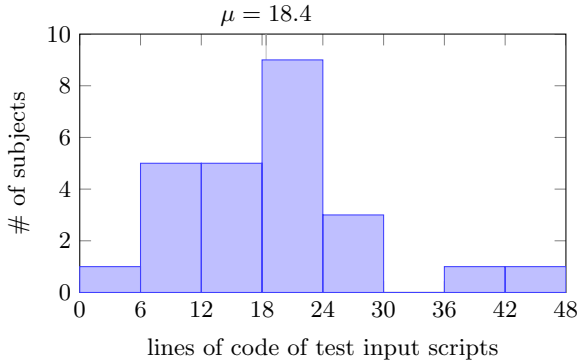

Figure 4: Histogram of LOC of test input scripts. Average LOC $\mu = 18.4$.

oper only needs to provide C^3 a test script (Section 4.1) that consists of (1) an initial software setup, (2) arguments to run the program, and (3) actions to be performed at runtime (for interactive programs only). All such efforts are contained in the test script. We show the statistics of test script LOC in Figure 4. Even if we are end-users of the evaluated subjects, writing such a short test script (5–43 with the mean of 18.4 lines of code) takes only a few minutes of work, which usually consumes less time than setting up the software from scratch. The longest test scripts are GNU make (simulating an incremental build) and GNU patch (generating patch files from two sets of synthesized files), which are 43 and 36 lines of code, respectively. Developers can also reuse their existing test cases by executing them in the test script.

5.2.3 Performance

We show the performance evaluation results in Table 3, which is conducted on a machine with limited computational power. For all evaluated subjects, C^3 finishes the entire process in minutes, which is certainly affordable in a testing environment.

There are two major factors that impact the performance of C^3: (1) taking file system snapshots to obtain an ES and (2) generation and validation of CSs. The CS generation and validation consumes the most of time but we still consider such cost is affordable because applications often interact with file system via limited patterns and a few test cases are sufficient to reveal potential crash consistency bugs. Due to the limitation of `ptrace`, C^3 cannot precisely decide which file system call is related to the virtual disk so ES are collected on all possible system calls. The results show that profiling slightly slows the program but this is only a minor issue because the slowdown is transparent to the program (as if the system call takes longer time to return). Profiling runs of GUI subjects take longer time than command-line subjects because we insert one-second delay between all consecutive GUI operations to ensure their completion.

6. LESSONS LEARNED

Handling file data demands caution. File system implementations usually do not provide a strong atomicity and persistence guarantee. Therefore, when user's contents are being erased (even if backup had been performed), the developer should be careful. Even experienced developers of mature software made mistakes (e.g., sort and gzip) and many "correct" solutions in our subjects (e.g., Vim and sed) are counter-intuitive or overkill.

Table 3: Performance evaluation results. Values in a row indicate #ES collected, #CS validated, profiling run time (denoted as P.), CS validation time (denoted as V.) and total time (including initial setup, three runs and CS validation), respectively. The last row displays averaged number of all 25 evaluated subjects.

Application	*Amount (#)*		*Time (minutes)*		
	ES	*CS*	*P.*	*V.*	*Tot.*
GNU make	710	277	0.09	0.89	1.19
GNU zip	726	647	0.01	1.85	1.89
bzip2	724	846	0.01	2.68	2.71
GNU coreutils sort	716	578	0.03	1.86	1.96
Perl	718	929	0.01	3.10	3.13
Shelve	754	211	0.01	0.65	0.67
Gimp	3,648	3,168	0.33	9.76	10.74
CuteMarkEd	2,216	483	0.31	1.46	2.41
TEXmaker	1,384	423	0.33	1.30	2.30
TEXstudio	2,258	937	0.33	2.86	3.84
Ted	804	859	0.02	2.63	2.71
jEdit	1,486	1,038	0.16	3.57	4.05
GitHub Atom	14,627	1,201	0.59	3.87	5.65
Adobe Brackets	7,197	1,704	0.59	5.53	7.28
Average (all subjects)	2,176	1,166	0.22	3.70	4.21

Even worse, protecting data safety is much trickier than it appears, because such data erasure may implicitly happen in the underlying libraries of which the developer may not be aware. For example, Python Pillow provides `image.save()` for writing an image, which opens the file with `O_TRUNC`. Using this function to change an image in-place is a crash consistency bug[9] and such a pattern is quite likely to occur in an image editing software[10]. Furthermore, developers tend to trust the crash consistency of a mature standard library, which also may not be valid. An example is Python standard library Shelve that uses GDBM as its backend by default, which does not provide any crash guarantee.

> Therefore, the rule of thumb is to handle file data with care (e.g., adding extra `flush` and `fsync` to ensure the persistence if performance is a secondary concern) or crash consistency should be validated with tools like C^3.

Library support matters. Relying on developers to handle all corner cases is impractical. It also seems impractical for libraries and file systems to provide strong consistency guarantee: file operations still are the bottleneck of many applications. Rather, libraries should provide means to protect data safety or explicitly document their behaviors or guarantee. Only well-designed libraries can relieve the developers' burden of considering file system crash behaviors.

Through the communications with the open-source community, we learned that many frameworks provide good solutions to safely manipulate files. For example, Qt provides `QSaveFile` and GTK provides `g_file_replace` to handle file operations "in the safest way possible". We also validated these two libraries by C^3 and we could not find an incon-

[9]We did not report this as a bug because Pillow offers no crash safety guarantee.

[10]The most famous open-source image editing software Gimp has a similar crash consistency bug in all versions before 2.9.

sistent crash site. Therefore, we strongly recommend developers to use libraries that explicitly document their safety (and such safety can be easily validated by C^3) in handling file data.

On the more emerging platforms, however, there lack crash-safe library support. In addition to the two aforementioned Python examples, Node.js as a server-side language only provides a simple library for file system operations. This design is valid because server programs usually persist data in a database. However, when the application domain of Node.js expands, such library becomes a weak link in crash consistency. GitHub Atom developers found the crash consistency bug difficult to fix because Node.js lacks a portable library for safely saving a document.

> Therefore, we recommend library developers to validate crash consistency of API use cases with tools like C^3 and explicitly document whether a library function provides crash consistency guarantee.

Crash consistency deserves more attention. A developer does not get rid of the crash consistency issue even if the software has nothing to do with the file system. Recall the GNU make example (Section 5.2.1) in which crash may lead to a corrupted objective file that has a up-to-date timestamp. It is not GNU make developer's responsibility to fix the problem; rather, it is GCC that does not meet crash-safety requirement of GNU make (timestamp should not be updated until output file is persisted). Yet, GCC as a compiler is not required to provide such guarantee. The cascading effect of a minor crash inconsistency finally leads to potentially severe consequences.

Even if GCC was crash-safe, GNU make allows arbitrary scripts to be executed in objective file generation and nobody can guarantee crash consistency for all of them. Therefore, the only solution is to alert users of such issues and start a build from scratch (e.g., by executing `make clean`) after a system crash.

> Finally, this example suggests that developers should receive more education on crash consistency. We believe that the results and analyses presented in this paper will be a wake-up call for general software developers to pay more attention to crash consistency.

7. RELATED WORK

Software reliability in various adverse conditions have been extensively studied. Examples are external system events that access conflicting resources [8], combination of multiple exceptional conditions [31], and reordered shared memory accesses in a relaxed memory model system [13].

This paper focuses on crash consistency, the particular adverse condition of system crash. As CPU and memory state vanish after crash, crash consistency bugs are scoped in the storage stack. The storage stack consists of layers of abstractions (hardware interface, device driver, file system, library, and database) and is finally used by the software.

At hardware level, the robustness of physical drives is studied [27, 33]. At file system level, data consistency and crash recovery are extensively studied [10, 14, 19, 26]. A file system implementation can be validated by testing [25] or model-checking [30].

However, even if the file system survives the crash, it does not guarantee atomicity and persistence of each individual file system call, leading to crash consistency bugs. Our previous work devised special workloads to exposure such bugs in databases [32]. Seminal work Alice [22] and eXplode [29] introduced general frameworks to validate crash consistency for both system software and applications, which largely inspired our work. Alice focuses on modeling crash behaviors across file system implementations and eXplode focuses on systematic exploration of execution paths that contain exceptional control-flow. However, such techniques are not sufficient to efficiently validate crash consistency of a wide varieties of applications. The generic oracle and test amplification in C^3 facilitate fully automated checking of crash consistency and they are orthogonal to the technical contributions of Alice and eXplode. One can integrate both the generic oracle and the test amplification of C^3 into Alice and/or eXplode. Furthermore, Alice depends on the abstract file system behavior model extracted from a profiling tool that may not be sound (leading to false positives), while any crash site reported by C^3 guarantees to be valid and can be manifested in practice. Recent work [9] studied crash consistency models, which resembles memory consistency models, to characterize and validate crash behavior of file systems. Checking an application's crash consistency against abstract file system models is a promising future direction.

An alternative approach to crash consistency is providing transaction among system calls, which can be achieved either by operating system support [17, 24, 28] or by hardware assistance [18, 20]. Finally, the ultimate solution to crash consistency is a file system implementation that has provable strong consistency guarantee (atomicity and persistence) for each individual file system call. Such possibility has recently been explored [11]. However, such work is still in its early stage to be realized in performance-critical production environments.

8. CONCLUSION

In this paper, we present C^3, a novel approach for validating the crash consistency of application software. The generic oracle and test amplification facilitate the automated validation of crash consistency for application software. Evaluation on real-world applications demonstrates the effectiveness and efficiency of C^3 in detecting crash consistency bugs.

We not only made C^3 public and open-source but also presented valuable lessons learned from the bugs discovered by C^3 and the communications with the open-source community. We hope the results in this paper will be a cornerstone for further enhancement of software reliability in terms of system crash.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for helpful comments and suggestions. This work was supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grant #61472177, #91318301, #61321491) of China, NSF grants #CCF-0953 759 (CAREER Award), #CCF-1319705, the CAS/SAFEA international Partnership Program for Creative Research Teams, China Scholarship Council (#201506190103), the program for Outstanding PhD candidate of Nanjing University, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

10. REFERENCES

[1] GDBM. http://www.gnu.org/software/gdbm/gdbm.html.

[2] Git. http://git-scm.com.

[3] LevelDB. https://code.google.com/p/leveldb.

[4] LMDB. http://symas.com/mdb/.

[5] Mercurial. http://mercurial-scm.org.

[6] SQLite. http://www.sqlite.org/.

[7] Linux kernel block driver docs, 2005.

[8] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 83–93, 2015.

[9] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 83–98, 2016.

[10] J. Carreira, R. Rodrigues, G. Candea, and R. Majumdar. Scalable testing of file system checkers. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, pages 239–252, 2012.

[11] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 18–37, 2015.

[12] E. C. R. Council. The economic impacts of the August 2003 blackout. 2004.

[13] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 244–254, 2010.

[14] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 307–320, 2007.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 72–81, 2013.

[17] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter. Improving server applications with system transactions. In *Proceedings of the ACM European Conference on Computer Systems*, EuroSys, pages 15–28, 2012.

[18] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu. High-performance and lightweight transaction support in flash-based ssds. *IEEE Transactions on Computers*, 64(10):2819–2832, 2015.

[19] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Ffsck: The fast file-system checker. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST, pages 1–16, 2013.

[20] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC, pages 221–234, 2015.

[21] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[22] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI, pages 433–448, 2014.

[23] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Crash consistency. *Communications of the ACM*, 58(10):46–51, 2015.

[24] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 161–176, 2009.

[25] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. Sibylfs: Formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 38–53, 2015.

[26] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[27] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the USENIX Conference on File and Storage Technologies*, volume 7 of *FAST*, pages 1–16, 2007.

[28] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the USENIX Conference on File and Storage Technologies*, volume 9 of *FAST*, pages 29–42, 2009.

[29] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI, pages 131–146, 2006.

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.

[31] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 595–605, 2012.

[32] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI, pages 449–464, 2014.

[33] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge. Understanding the robustness of SSDs under power fault. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST, pages 271–284, 2013.

APPENDIX

A. INTRACTABILITY RESULTS

THEOREM 1. *Calculating editing distance (defined in Section 3.3) is NP-Complete.*

PROOF. (sketch) We prove the NP-Completeness by a reduction from the bin packing problem. The bin packing problem decides the possibility of packing n indivisible items (each weights a_i for $1 \leq i \leq n$) into k bins (each weights w and $\sum a_i = k \cdot w$). Bin packing is NP-Complete, even for the special case of $k = 2$ (partition problem) [15].

We reduce an arbitrary bin packing instance ($a_1, \ldots, a_n$, k and w) to an editing distance instance constructed as follows: let the CS consists of n files named $x_1, \ldots, x_n$, each contains a_i bytes of zero. Let the ES consists of k files named $y_1, \ldots, y_k$, each contains w bytes of zero. The bin packing-problem has a solution if and only if the editing distance from the CS to the ES is $2n - k$.

To transform the CS to the ES, n meta-data operations are required: k renamings (rename from x to y) and $n - k$ deletions (redundant files). Furthermore, contents in the $n - k$ deleted files should be moved to the k renamed files. This gives a lower-bound of editing distance to be $2n - k$.

If the editing distance is exactly $2n - k$, each file performs at most one content move operation (and moves its all contents), which simulates the bin packing procedure to obtain a valid solution. On the other hand, if there is a bin packing solution, we simply move file contents in each of the $n - k$ files to their destinations, yielding an editing distance of $2n - k$, completing the proof. □

Finally, one might argue that computing the exact value of editing distance is overkill and any approximate or heuristic algorithm may work well in practice. However, note that contents in a file system snapshot can be many (e.g., $10^7 \approx$ 10M bytes of data), it would be extremely difficult to scale such an algorithm that typically adopts dynamic programming or linear programming related technique.