CrashTalk: Automated Generation of Precise, Human Readable, Descriptions of Software Security Bugs

Kedrian James
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA

Kevin Valakuzhy
Georgia Institute of Technology
Atlanta, GA, USA

Kevin Snow
Zeropoint Dynamics
Chapel Hill, NC, USA

Fabian Monrose
Georgia Institute of Technology
Atlanta, GA, USA

ABSTRACT

Understanding the cause, consequences, and severity of a security bug are critical facets of the overall bug triaging and remediation process. Unfortunately, diagnosing failures is often a laborious process that requires developers to expend significant time and effort. While solutions have been proposed to help expedite the process of pinpointing the cause of a security bug, few proposals provide an *explanation* along with a diagnosis to make the bug discovery and triaging process less taxing. Moreover, even in cases where descriptions are provided, they are not guided by classification models that support precise descriptions of the flaw.

We present an approach that uses static and dynamic analysis techniques to automatically infer the cause and consequences of a software crash and present diagnostic information following NIST's recently released Bugs Framework taxonomy. Specifically, starting from a crash, we generate a detailed and accessible English description of the failure along with its weakness types and severity, thereby easing the burden on developers and security analysts alike. To evaluate the effectiveness of our approach, we compare our ability to find fault locations and generate explanations compared to that of professional software developers by using a benchmark specifically designed to assist with realistic evaluation of tools in software engineering. In addition, using 33 real-world vulnerabilities we collected, we show that our approach correctly diagnoses over 94% of the failures and, in some cases, generates weakness types that are more specific than those that were originally assigned by the submitter or National Vulnerability Database analysts. We also generate initial vulnerability scores that can be used by project managers to assist with prioritizing bug fixes. On average, the overall process takes just over a minute, which is orders of magnitude faster than what professional developers can do.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Bug Localization; Debugging; Vulnerabilities

ACM Reference Format:
Kedrian James, Kevin Valakuzhy, Kevin Snow, and Fabian Monrose. 2024. CrashTalk: Automated Generation of Precise, Human Readable, Descriptions of Software Security Bugs. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24), June 19–21, 2024, Porto, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626232.3653256

1 INTRODUCTION

Due to the negative implications of software breaches, many companies and open-sourced projects have resorted to automated software testing techniques (*e.g.,* fuzz testing) to discover vulnerabilities before adversaries are able to exploit them. That movement has led to a significant increase in the number of bugs [30] reported over the past decade. Yet, despite these successes, considerable delays between discovery and fixing times persist. In particular, developers often lament about spending an exorbitant amount of time [71, 72] analyzing bug reports to identify and understand the bug responsible for the failure, independent of the time it takes to assess the severity of the failure and to fix it.

To expedite bug fixing, a good bug report should contain an overview of the failure along with its location and a detailed diagnosis of the bug causing the failure. In practice, however, bug reports widely vary in quality, based on the level of expertise of the bug hunter and the tools they use [9, 85]. Although several approaches have been proposed to diagnose and locate the cause of a failure, Hirsch and Hofer [34] found that contemporary solutions often fall short in meeting the needs [40] of developers because the output of these tools lacks critical information, the content is ambiguous, and/or the output provides insufficient means to understand the severity of the vulnerability. In fact, a recent study of 250 popular projects on GitHub revealed that, on average, 70% of bug reports lacked content needed to successfully fix the bug [64]. Equally troublesome is the fact that many approaches rely on the description of a vulnerability to predict its severity, but it is not uncommon for imprecise descriptions to lead to improper bug prioritization [18, 19].

But there is good news. Several studies [9, 64, 85] examining the quality of bug reports have shown that reports that are easier to read and contain helpful information (*e.g.,* cause, consequence, and severity) are more likely to get fixed. The dire need for better diagnostic tools was highlighted in a recent mixed-method study [75] of 386 bug finders and developers that found that "understandability/readability of fixes is a key concern for developers and something that should be taken into consideration". To fill that void, we propose a pragmatic solution that automatically ① diagnoses the cause of a crash and uses the taxonomy from NIST's Bugs Framework [14–16] to explain the flaw, ② assigns relevant weakness types (CWEs) to the failure, and ③ assigns the severity of the failure using the Common Vulnerability Scoring System (CVSS). Our specific contributions include:

- We provide an approach, called `CrashTalk`, that uses static and dynamic analysis techniques to collect artifacts that are helpful for understanding the cause of a security bug.
- We use artifacts that map to components expressed in the Bugs Framework to generate reports that succinctly describe the reasons for the failure. The framework is a descriptive model, and to our knowledge, this is the first end-to-end implementation of a tool that takes in a binary and associated crashing input and outputs a concise description of the flaw.
- We leverage the components of the Bugs Framework along with other characteristics of the failure to automatically assign weakness types (CWEs) to the failure, and assign severity rankings based on the Common Vulnerability Scoring System (CVSS).
- We perform evaluations (on widely used synthetic data, as well as on real-world examples we curated from 33 bugs) to demonstrate how well our approach can explain the cause of a failure. In addition, we perform evaluations using a human-generated benchmark for the qualitative evaluation of automated fault localization, bug diagnosis, and repair techniques [12].

2 BACKGROUND

Triaging is the process of analyzing software crash reports. The triaging process typically involves deduplication, prioritization, and assignment. During the deduplication phase, a triager inspects crash reports to identify and group duplicates. After deduplication, each unique crash report is analyzed to determine the underlying bug that causes the crash, after which the bug is assigned a priority based on its severity (*i.e.,* potential damage the flaw could cause) [35, 39, 69]. While all stages of the triaging process are important, prior studies [24, 34, 40] have found that localizing a bug and explaining its cause is one of the most challenging tasks, especially for memory-based bugs. As such, developers often heavily rely on a tool's output to help diagnose and fix a bug.

2.1 Describing Security Bugs

One of the most widely adopted approaches for describing and classifying bugs is the Common Weakness Enumeration (CWE) system. The CWE is a community-developed list of software and hardware weaknesses along with their description and mitigation strategies [46]. While helpful to the security community at large, the CWE is a hierarchical structure with several interdependencies among the weakness types, many of which are broad and ambiguous [22, 76]. Furthermore, the NIST software assurance group found that, on its own, the CWE classification is not sufficient, accurate, and precise enough to serve as the common language for describing software bugs [14, 76].

An effort to improve the state of the practice is the Bugs Framework (BF) [14–16] by NIST that provides a taxonomic model for describing software bugs. In essence, it is a structured extension of the CWE system, facilitating bug-reporting tools to generate more precise descriptions of software bugs and vulnerabilities.

The Bugs Framework consists of bug classes that represent specific phases of the execution of a program. Each class is comprised of a set of operations (*e.g.,* read or write) that are necessary for the specific phase of execution, the operands for the specific phase (*e.g.,* memory object), a set of attributes that describe the operation and operands, a valid cause-consequence relationship, and sites to describe locations in the source code where a bug can occur [13]. To facilitate precise causal descriptions, each weakness is expressed using one cause, one operation, and one consequence. For instance, applying concepts from the Bugs Framework Memory Model [15] one could track the state of an object (*i.e.,* memory buffer) from allocation to deallocation, including the operations performed on the buffer. To help with the labeling of new bugs, NIST released a GUI-based tool that lets a user manually specify vulnerabilities using the Bugs Framework language.

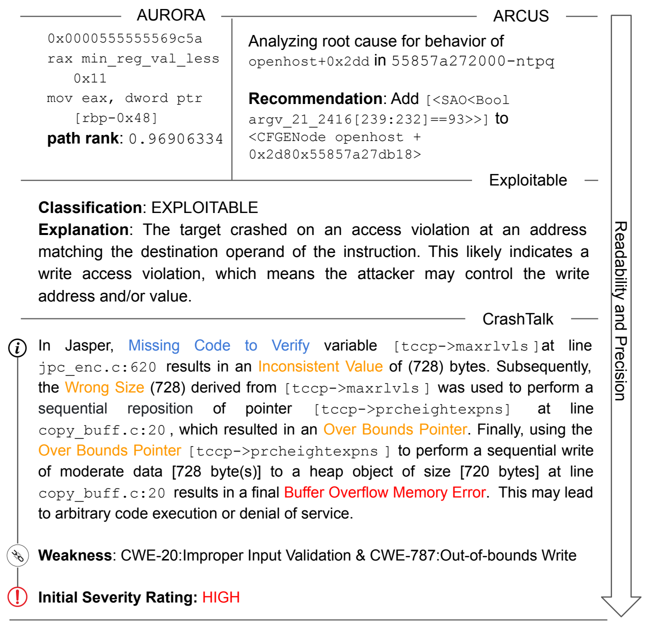
In Section 3, we describe how we instrument memory and extract the information needed to auto-generate reports with succinct descriptions of the cause and consequences, as well as infer the weakness types and severity of a bug.

2.2 Measuring the Quality of Bug Reports

A key factor when assessing the quality of bug reports is readability. This is supported by the observation of Zimmermann et al. [85] that bug reports that are easier to read appear to have shorter lifespans than their counterparts. A commonly used readability metric is the Flesch Reading Ease score [38, 45], which uses the average number of words per sentence and the average number of syllables per word to approximate readability.

We use an extension known as the Flesch-Kindcaid Grade Level Score that is more suitable for technical texts, and seeks to capture complexity as well as simplicity of the prose. A higher score indicates that the text includes more specialized vocabulary (e.g., on exploitation and memory errors) versus less technical content that may be easier to read by someone at a lower grade level.

Besides measuring the readability of a bug report, it is also important to ensure that the description of the failure is specific enough for developers and security analysts to understand the weakness classes associated with a failure, allowing them to better analyze its impact and devise effective mitigation strategies [13, 22]. As a best practice, NIST and MITRE recommend mapping vulnerabilities to the most specific weakness [47, 52]. To better facilitate the assignment process, analysts with the National Vulnerability Database (NVD) use a subset of the overall CWEs, known as a slice, that best represents the distribution of the overall CWEs [52]. The slice contains both class and base CWE types. The class CWE types provide a general description of the CWE, while the base CWEs provide more specific descriptions. Since the BF is a structured extension of the CWE classification system designed to provide precise descriptions of bugs and vulnerabilities, CWEs can be assigned


AURORA
ARCUS
Exploitable
CrashTalk
Readability and Precision

```
0x0000555555569c5a
rax min_reg_val_less
   0x11
mov eax, dword ptr
   [rbp-0x48]
```
path rank: 0.96906334

Analyzing root cause for behavior of `openhost+0x2dd` in `55857a272000-ntpq`

Recommendation: Add `[<SAO<Bool argv_21_2416[239:232]==93>>]` to `<CFGENode openhost + 0x2d80x55857a27db18>`

Classification: EXPLOITABLE
Explanation: The target crashed on an access violation at an address matching the destination operand of the instruction. This likely indicates a write access violation, which means the attacker may control the write address and/or value.

In Jasper, Missing Code to Verify variable `[tccp->maxrlvls ]` at line `jpc_enc.c:620` results in an Inconsistent Value of (728) bytes. Subsequently, the Wrong Size (728) derived from `[tccp->maxrlvls ]` was used to perform a sequential reposition of pointer `[tccp->prcheightexpns]` at line `copy_buff.c:20`, which resulted in an Over Bounds Pointer. Finally, using the Over Bounds Pointer `[tccp->prcheightexpns ]` to perform a sequential write of moderate data [728 byte(s)] to a heap object of size [720 bytes] at line `copy_buff.c:20` results in a final Buffer Overflow Memory Error. This may lead to arbitrary code execution or denial of service.

Weakness: CWE-20:Improper Input Validation & CWE-787:Out-of-bounds Write

Initial Severity Rating: HIGH

Figure 1: CrashTalk's (our approach) ***automatically*** **generated report versus other tools.**

to specific bug classes based on their consequences. To assess the level of specificity in a bug report, we use the components (*e.g.,* cause, consequences, operation) to extract and assign appropriate weakness types to a failure. In Section 3, we provide details on the extraction and assignment process.

Unfortunately, we found that reports generated by contemporary tools and existing research prototypes pay little attention to readability and specificity. To address this gap, we focus on auto-generating high-quality reports by evaluating their readability and specificity. Figure 1 shows the explanations from the popular `exploitable` [28] tool, the most closely-related research prototypes AURORA [11] and ARCUS [81], and our approach.

2.3 Evaluating the Severity of the Failure

One of the most widely adopted approaches for evaluating the severity of a vulnerability is the Common Vulnerability Scoring System (CVSS), which consists of three metric groups: the Base, Temporal, and Environmental. The Base metric group describes the intrinsic characteristics of a vulnerability that remain constant irrespective of time or user environments [18, 19, 26]. In contrast, the Temporal metrics capture attributes of the vulnerability that may change over time but not across user environments. The Environmental Metrics serve to adjust the severities identified in both the Base and Temporal groups based on specific environmental factors [26]. For our approach, we compute the severity of a failure using the components expressed in the Base metric group. In Section 3, we provide details on how we use the components of the Bugs Framework along with other information to evaluate the severity of the bug.

3 APPROACH

Operationally, our approach consists of the three phases shown in Figure 2. The emulation phase ① takes a crashing input and associated program and performs emulation. The crash analysis phase ② uses the data collected from the program emulation along with source code and debugging information to pinpoint the cause of a crash. Next, the report generation phase ③ uses information from the analysis phase together with the Bugs Framework model [13] to explain the steps leading to the flaw (3A). Afterward, component 3B computes the readability of the report, and if it is above a predefined threshold, the report is presented to the user.

3.1 Phase ① Emulation

Given a buggy program along with the crashing input, we first reproduce the bug in an emulated setting. We expand on the data collected by other approaches to provide an annotated execution trace that can be used to determine the source of values that lead to crashes. Our approach is conceptually similar to the dynamic data flow tracking done by Kemerlis et al. [37] (coined `Libdft`) in that we construct data dependencies by recording reads and writes of memory and registers dynamically, from both instructions and system calls. Dynamic data flow tracking avoids the challenges of points-to analysis or reverse execution faced by static data flow approaches. However, rather than propagating taint, we track concrete values that are written to registers and memory, as well as the assembly instruction responsible for assigning that value, which we call the "source" of the value. Determining the source of a value cannot be done by simply tracking changes in the register or memory values because overwriting a value with an identical value changes its source. Thus, we need a way to infer the registers that an instruction will modify.

To determine the register dependencies of instructions in an architecture-agnostic way, we convert each assembly-level instruction to QEMU's Tiny Code Generator (TCG) [8] Intermediate Representation (IR). Our decision to use TCG stems from the fact that QEMU is widely supported and allows us to generate TCG for any computer architecture it can emulate. Furthermore, the stability and accuracy of QEMU's emulation provides a good level of confidence in the accuracy of the IR it generates.

Additionally, to distinguish between separate executions of an instruction at the same virtual address, we store data dependencies relative to the complete instruction trace. In doing so, we attain full context-sensitivity for our analysis, allowing us to distinguish exact dependencies even across function boundaries. At present, we have implemented a set of data flow rules for a subset of TCG instructions that cover the most common instructions (e.g., arithmetic and logical instructions). Our architecture is modular, allowing us to add new rules on a case-by-case basis (i.e., whenever we encounter unsupported instructions while diagnosing the reasons why a program under scrutiny crashed).

To deal with the fact that certain types of memory corruption bugs (*e.g.,* heap-buffer overflows, heap use-after-free) may only produce a crash in the presence of memory sanitizers, we also perform memory annotation during emulation. We found it simpler to implement memory sanitization techniques in the emulation framework than to integrate existing sanitizers. Specifically, we

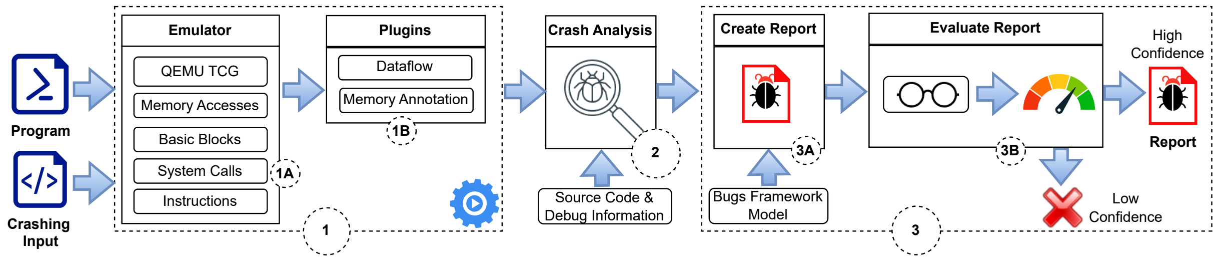


Figure 2: Overall workflow of `CrashTalk`: the emulation phase ①, the crash analysis phase ②, and the report generation phase ③. Part `3A` uses the Bugs Framework to generate the report. Part `3B` checks the readability of the report to ensure it is above a user-defined threshold.

implement guard pages along with fine-grained memory tracking to force a crash whenever certain conditions are violated. To detect heap-buffer overflows, we first hook calls to heap memory allocation routines (*e.g.,* malloc, calloc). We allocate the requested memory along with additional memory to create a guard buffer adjacent to the requested memory that will detect read or write accesses outside the bounds of the allocated memory and trigger a crash. Similarly, to detect heap use-after-frees we hook the memory deallocation routine `free` to keep track of freed memory buffers. We also implement a guard in the freed memory space to trigger a crash upon read and write accesses. For double-frees, we force a crash if the memory passed to the `free` function was previously freed. Similarly, we force a crash for invalid frees if the memory passed to the `free` function is not previously allocated.

Our sanitizer can track byte-level access of protected regions. This fine-grained memory tracking allows us to keep track of operations (*e.g.,* read or write), and instructions that operate on each allocated heap buffer. We can also customize the size of guard data. In our implementation, we chose to follow each heap allocation (call to malloc, calloc, or realloc) with a 16-byte guard buffer that will identify invalid memory accesses within 16 bytes of the end of the memory region. A shortcoming is that we can miss invalid accesses that are more than 16 bytes away from a valid memory region. For a more robust invalid memory detection mechanism, we can increase the size of the guard buffers at the cost of increasing the memory overhead of the memory sanitization module.

3.2 Phase ② Analysis

Whenever the program crashes during emulation, the information available at the time of the crash is passed to this phase along with the program source code and the debug information. The goal is to understand the cause of the crash.

To do so, we construct a backward taint graph starting with the address of the crashing instruction. Using the information recorded during emulation, we identify the values that are read by that instruction and used in the operation that resulted in the crash. We sift through the recorded execution trace to find the sources of those values, drawing a data dependency between the source and use. The objective is to construct definitive data dependencies between two instructions, as opposed to dependencies produced by context-insensitive data flow algorithms that only produce a set of possible candidates for data dependencies. That said, not all data dependencies represent meaningful connections that are useful for providing context to crashes. A common source of spurious data dependencies is instructions that reference values on the stack. Considering the stack pointer as a data dependency of a value on the stack would be problematic as our approach would then consider all modifications of the stack pointer (including all prior push and pop instructions) as data dependencies. This problem is similar to the taint explosion issue commonly experienced by taint analysis techniques. In our case, we avoid these spurious connections by ignoring all data dependencies involving registers containing stack or base pointers, specifically ESP, EBP, RSP, and RBP. Yet another source of spurious data dependencies is implicit data dependencies, or data dependencies involving control flow instructions. In our current prototype, we do not consider implicit data dependencies.

Given the captured data dependencies, we perform memory state tracking to pinpoint the source of the value that caused the crash. We extract the value that caused the violation along with the corresponding memory address/register from the crash site. We use the graph to identify where that invalid value was defined and label that point as a possible cause depending on the type of bug (*e.g.,* null-pointer-dereference). We annotate the graph by mapping the assembly-level instructions to their corresponding line numbers in the source code by parsing the program debugging information. To enhance the readability of our generated reports, we also annotate the graph with the variable names that correspond to each memory address, information that is also available from program debug symbols. The annotated data-flow graph, along with the guard page violation information, is sent to the next stage in the pipeline.

3.3 Phase ③ Report Generation

The final step uses the aforementioned data to auto-generate a report. To illustrate how this process works, we use an example, `CVE-2020-27828`, that is a heap-based buffer overflow in the JasPer image manipulation library that arises because of a missing verification for a user-specified maximum number of resolutions for an image. NVD last updated its entry for this bug on 11/06/2023, but their description still does not offer the level of detail we provide.

Motivating Example: In Figure 3 at `line 2a`, heap memory is allocated for a coding parameter structure `cp`, which also contains a per-title coding parameter structure `tccp`. The `tccp` structure includes a buffer `prcheightexpns` of size 33, and an unsigned integer `maxrlvls` representing the maximum resolution level. Later in the

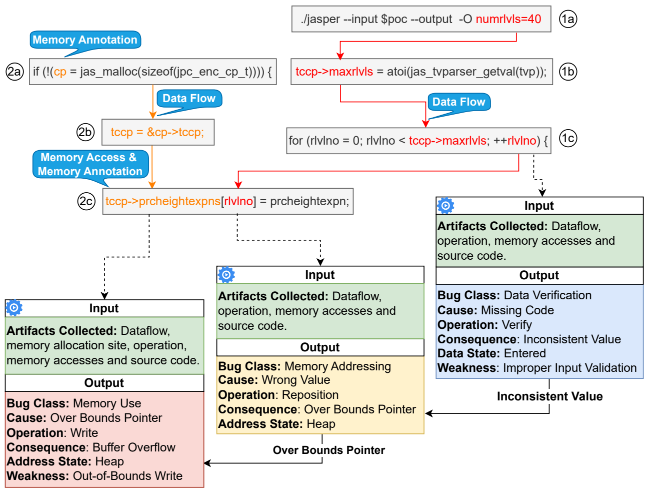

./jasper --input $poc --output -O numrlvls=40
1a
Memory Annotation
2a
if (!(cp = jas_malloc(sizeof(jpc_enc_cp_t)))) {
tccp->maxrlvls = atoi(jas_tvparser_getval(tvp));
1b
Data Flow
Data Flow
2b
tccp = &cp->tccp;
for (rlvlno = 0; rlvlno < tccp->maxrlvls; ++rlvlno) {
1c
Memory Access & Memory Annotation
2c
tccp->prcheightexpns[rlvlno] = prcheightexpn;
Input
Artifacts Collected: Dataflow, operation, memory accesses and source code.
Output
Bug Class: Data Verification
Cause: Missing Code
Operation: Verify
Consequence: Inconsistent Value
Data State: Entered
Weakness: Improper Input Validation
Inconsistent Value
Input
Artifacts Collected: Dataflow, operation, memory accesses and source code.
Output
Bug Class: Memory Addressing
Cause: Wrong Value
Operation: Reposition
Consequence: Over Bounds Pointer
Address State: Heap
Over Bounds Pointer
Input
Artifacts Collected: Dataflow, memory allocation site, operation, memory accesses and source code.
Output
Bug Class: Memory Use
Cause: Over Bounds Pointer
Operation: Write
Consequence: Buffer Overflow
Address State: Heap
Weakness: Out-of-Bounds Write

Figure 3: Dataflow graph containing relevant source lines for a heap-based buffer overflow vulnerability in JasPer (CVE-2020-27828). The blue labels denote that we use our dataflow, memory access, and annotation modules to collect information at that stage.

program at `line 1b`, `tccp->maxrlvls` is assigned a value specified by the user (*i.e.,* 40 for this specific bug). Subsequently, `maxrlvls` is used at `line 1c` as the upper bound for the loop. However, there is no verification between the user-specified `maxrlvls` at `line 1c` and the size of the `prcheightexpns` buffer. As a result, the program crashes at `line 2c` with an out-of-bounds write. This failure maps to two improper states, namely (i) when `prcheightexpns` is positioned above its upper bounds and (ii) when data is written to `prcheightexpns` beyond its upper bounds. In the BF, these improper states correspond to the memory addressing (MAD) bug class, and memory use (MUS) bug classes, respectively.

Based on the improper state related to the MUS class we mark the action of writing data to the heap using an out-of-bounds pointer as the *cause* and the heap-buffer overflow as a *consequence* of this state (see red rectangle in Figure 3). We use the information provided by the memory annotator along with the source code and debug information to extract the prerequisite BF components (*e.g.,* operation, address state). Following that procedure, we use components of the MAD bug class (see yellow rectangle in Figure 3) to describe how the previous state came to be. Specifically, we follow the dataflow and parse the source code to determine the operation (i.e., a loop at line 1c) used to sequentially reposition `prcheightexpns`. Since `tccp->prcheightexpns` was positioned over its bounds, we attribute the cause to a `wrong size` for repositioning, and the consequence is the over-bounds pointer `prcheightexpns`. Next, we check to determine if the overflow resulted from an incorrect memory allocation (*e.g.,* integer overflow). Since this is not the case in this scenario, we default to the data verification bug class to signify a missing verification between `tccp->maxrlvls` and the size of `tccp->prcheightexpns`.

Weakness Types: Our selection of CWE weakness types is guided by the information provided by both Mitre [47] and the NVD [52]. We assign CWEs for each type of failure from the NVD slice [52]. The type and description of each CWE we select are determined by specific failure types identified by our analysis component (e.g., heap-based buffer overflow), the corresponding operation (read or write) that led to the failure, the memory space (e.g., stack vs. heap) involved in the failure, and the components of the Bugs Framework (e.g., cause, consequence, and attributes). For instance, in the motivating example, our approach would designate *Improper Input Validation* (*e.g.,* CWE-20) as the initial weakness type because the corresponding Bugs Framework bug class is Data Verification, and the cause was a missing verification for the entered data. Similarly, we would identify the final consequence as an *out-of-bounds write* (*e.g.,* CWE-787) based on the fact that data was written beyond its bounds. Furthermore, we explicitly specify that the write occurred in the heap by examining the memory space of the crash.

Severity: To assess the severity of a failure, we utilize the weakness types, components of the Bugs Framework, and other information we extracted during program emulation (*e.g.,* system calls, use-def chain). We then use the scoring rubric for the CVSS V3.1 [26, 27] to formulate rules and assign values to the base metric components. Because the *Attack Complexity* and *Scope* components are inherently subjective and require human analysis, we assign default values as outlined in the CVSS user guide [27]. The *Attack Vector* component determines how the vulnerable program can be exploited (*e.g.,* over a network or via a local application). To determine this value, we analyze the data flow and system calls to determine whether the program is bound to the network stack or operates locally. The *User Interaction* component denotes if the attack requires a user to perform a specific action, such as opening a specifically crafted file. To determine if this criterion is met, we again analyze the dataflow to decide if the program accepted data either from the network or locally. To determine the impacts on Confidentiality, Integrity, and Availability, we utilize the inferred weakness types along with extracted causes, consequences, and attributes of the failure.

For instance, in Figure 3, the program has a high impact on availability since the out-of-bounds write causes the program to crash. Additionally, the attacker controls how much data is written out-of-bound, so the impact on Integrity in this case would also be high. Further, since the program requires a specially crafted file as input, the User Interaction field would be set to required, and the Attack Vector would be set to network because the program is a library that can be used in networked programs. Once our run-time system has collected all the information needed to populate values for each field, we use the *CVSS 3.1* equation [26] to compute the base score metrics and assign a severity level.

3.4 Implementation Details

`CrashTalk` is implemented utilizing Python-based binary emulation as its foundation. We extended an open-sourced emulation engine, called `zelos` [1], to replicate the bug within a simulated environment. In our context, the functionality of `zelos` resembles that of the widely employed `qemu-user-mode` tool in crash analyses. However, a notable distinction lies in the extensive API provided by `zelos` for dynamic code and memory instrumentation and analysis. Unlike the latter, `zelos` emulates system calls instead of forwarding them to the underlying operating system. This feature facilitates easy and accurate tracing of dataflows across the user-kernel boundary. To fulfill our objectives, we also developed plugins for dataflow analysis and memory annotation, as illustrated by the components depicted in Figure 2. We note that alternative solutions for dataflow tracking exist, such as dynamic binary instrumentation (e.g., Intel PIN, DynamoRio, `libdft`), comprehensive emulated system dataflow tracking (e.g., `panda.re`), or other userspace emulators (e.g., `angr`, `qiling`, `qemu-user-mode`). However, each of these options presents different limitations, such as performance, compatibility, or accuracy of flow tracking, which diminish their practicality for our task.

We used the `pyelftools` Python API to parse the DWARF information, and the textstat Python API to compute the readability scores for the reports. The average elapsed time for the emulation, dataflow reconstruction, and analysis phases for processing the 33 real-world bugs we analyze in §4 are 38, 2.7, and 22.8 seconds respectively.

4 EVALUATION

To evaluate the effectiveness of our approach, we designed experiments to answer (*RQ1*): can we accurately collect the necessary details needed as input for the Bugs Framework, and (*RQ2*): how well do our explanations align with diagnoses provided for real-world bugs?

4.1 RQ1: On Collecting Prerequisite Information

For our empirical evaluations regarding RQ1, we use a dataset with 2,942 test cases from the Juliet 1.3 suite [10]. We use the Juliet 1.3 suite because it was specifically designed to evaluate automated security tools and contains many test cases for each final error along with the ground truth [10]. That dataset has been widely used in security evaluations (*e.g.,* [31, 44, 74, 81]). Each test case contains both a flawed (*i.e.,* positive) and a non-flawed (*i.e.,* negative) version. For each final error type, we randomly select a set of test cases that cover the possible flow variants defined in the test suite.

Final Error	Precision	Recall	F1 Score
Double-Free	1.00	0.98	0.99
Stack-Buffer Overflow	1.00	1.00	1.00
Heap-Buffer Overflow	1.00	1.00	1.00
Use-After-Free	1.00	0.97	0.98
NULL-Pointer-Dereference	1.00	0.91	0.95
Average	**1.00**	**0.97**	**0.98**

Table 1: Accuracy of failure type identification on the Juliet Dataset.

Specifically, our objective is to determine if we can accurately identify the final errors (*e.g.,* use-after-free) and the components of the Bugs Framework (*e.g.,* cause, consequence, attributes, and sites) associated with the failure. To conduct the experiment, we compile each flawed and non-flawed test case individually and then execute them within our emulated environment. We run each experiment 10 times and report the average scores. In all cases in the Juliet dataset, we can consistently identify the final errors with no false positives. Table 1 shows that in addition to locating the sites, we are also able to successfully extract the live variables corresponding to the cause and consequences. In a few cases, we are not able to reproduce a crash, resulting in false negatives.

Final Error	Precision	Recall	F1 Score
Bug Class, Operation, Cause	1.00	0.95	0.97
Consequence, Mechanism, Source Code	1.00	0.95	0.97
Site, Address State, Size Kind	1.00	0.95	0.97
Average	**1.00**	**0.95**	**0.97**

Table 2: Accuracy of BF component identification on the Juliet Dataset.

Although the Juliet 1.3 test suite has been widely adopted for evaluating security tools, the test cases are much simpler than actual bugs found in real-world programs. This is because the test suite contains artificially generated code designed to simplify the evaluation process. As a consequence, the results achieved on the Juliet test suite may not reflect how well a technique might perform on more complex real-world programs [31]. To evaluate our approach in more realistic settings, we turn to real-world programs.

4.2 RQ2: On the Alignment with Known Diagnoses of Real-world Errors

To address the fact that the Juliet dataset may not be representative of flaws in real programs, we conduct experiments on two datasets. First, we use the `DBGBench` benchmark introduced by Böhme et al. [12] which is designed to help with the evaluation of software engineering tools. `DBGBench` is a human generated benchmark that includes real errors found in open-sourced C projects alongside data from debugging sessions with 12 professional software engineers. Böhme et al. [12] show that in the absence of user studies, the information that was painstakingly collected in their benchmark allows for in-depth evaluations that are grounded in practice, thereby minimizing a number of potentially unrealistic assumptions that could be made by tool designers. In particular, they argue that "`DBGBench` can be used to evaluate *without a user study* how well novel automated tools perform against professional software developers in the tasks of fault localization, debugging, and repair." To that end, Böhme et al. [12] suggest that the output of the tool under investigation should be compared to the consolidated error diagnoses derived by human experts.

	Developer Diagnosis		Our Approach	
Bug ID	Time (Min)	Explanation	Time (Min)	Diagnosis
find.07b941b1	23.7	Slightly Difficult	0.48	●
find.93623752	50.8	Moderately Difficult	0.42	●
find.c8491c11	31.4	Slightly Difficult	0.41	●
find.dbcb10e9	22.9	Slightly Difficult	0.46	●
grep.3220317a	63.7	Moderately Difficult	-	-
grep.3c3bdace	67.6	Very Difficult	0.38	●
grep.54d55bba	26.7	Slightly Difficult	0.47	●

Table 3: Crash bugs [12] used. The ● symbol denotes a correct diagnosis.

To conduct such an analysis, we evaluated 7 crashing bugs from `DBGBench` that we could recreate.[1] For each bug, we followed the

[1]Although the dataset suggests other bugs, `find.93623752` included a crashing input but is not labeled as a crash bug. `find.24bf33c0` appears to be mislabeled. In the case of `find.091557f6`, we were unable to reproduce the crash described in the report.

		Actual			Alignment			
Program	**CVE**	**Failure**	**Severity**	**Assigned CWE**	**Failure**	**Cause**	**Severity**	**CWE**
binutils	CVE-2006-2362	stack-buffer-overflow	v2:High	NIST NVD-CWE-Other	●	●	v3:High	**Improper Input Validation Out-of-bounds Write**
binutils	CVE-2017-6966	heap-use-after-free	v3:Medium	Use After Free	●	●	↑	Use After Free
binutils	CVE-2018-20623	heap-use-after-free	v3:Medium	Use After Free	●	●	↑	Use After Free
binutils	CVE-2019-9077	heap-buffer-overflow	v3:High	Out-of-bounds Write	●	●	●	Improper Input Validation Out-of-bounds Write
jasper	CVE-2011-4516	heap-buffer-overflow	v2:Medium	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	●	v3:High	**Improper Input Validation Out-of-bounds Write**
jasper	CVE-2011-4517	heap-buffer-overflow	v2:Medium	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	●	v3:High	**Incorrect Calculation of Buffer Size Out-of-bounds Write**
jasper	CVE-2020-27828	heap-buffer-overflow	v3:High	Improper Input Validation	●	●	●	Improper Input Validation Out-of-bounds Write
libjpeg-turbo	CVE-2012-2806	heap-buffer-overflow	v2:Medium	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	●	v3:High	**Improper Input Validation Out-of-bounds Write**
libjpeg-turbo	CVE-2017-15232	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
libjpeg-turbo	CVE-2018-14498	heap-buffer-overflow	v3:Medium	Out-of-bounds Read	●	●	●	Improper Input Validation Out-of-bounds Read
libjpeg-turbo	CVE-2018-19664	heap-buffer-overflow	v3:Medium	Out-of-bounds Read	●	●	●	Improper Input Validation Out-of-bounds Read
libjpeg-turbo	CVE-2020-13790	heap-buffer-overflow	v3:High	Out-of-bounds Read	●	●	●	Improper Input Validation Out-of-bounds Read
libming	CVE-2018-6358	heap-buffer-overflow	v3:High	Out-of-bounds Write	●	●	●	Improper Input Validation Out-of-bounds Write
libtiff	CVE-2015-8668	heap-buffer-overflow	v3:Critical	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	●	↓	**Improper Input Validation Out-of-bounds Read**
libtiff	CVE-2017-9117	heap-buffer-overflow	v3:Critical	Out-of-bounds Read	●	●	↓	Improper Input Validation Out-of-bounds Read
libxml2	CVE-2015-7498	heap-buffer-overflow	v2:Medium	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	○	○	○
libxml2	CVE-2016-1835	heap-use-after-free	v3:High	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	●	●	Use After Free
libxml2	CVE-2017-5969	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
libxml2	CVE-2017-9049	heap-buffer-overflow	v3:High	Out-of-bounds Read	●	○	○	○
libzip	CVE-2017-12858	double-free	v3:Critical	Double Free	●	●	↓	Double Free
nasm	CVE-2004-1287	stack-buffer-overflow	v2:High	NIST NVD-CWE-Other	●	●	v3:High	**Improper Input Validation Out-of-bounds Write**
nasm	CVE-2017-10686	heap-use-after-free	v3:High	Use After Free	●	●	●	Use After Free
nasm	CVE-2018-16517	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
nasm	CVE-2022-44370	heap-buffer-overflow	v3:High	Out-of-bounds Write	●	●	●	Improper Input Validation Out-of-bounds Write
openjpeg	CVE-2016-7445	null-pointer-dereference	v3:High	NULL Pointer Dereference	●	●	↓	NULL Pointer Dereference
openjpeg	CVE-2016-10505	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
openjpeg	CVE-2021-3575	heap-buffer-overflow	v3:High	Out-of-bounds Write	●	●	●	Improper Input Validation Out-of-bounds Write
pcre	CVE-2015-3210	heap-buffer-overflow	v3:Critical	Improper Restriction of Operations within the Bounds of a Memory Buffer	●	●	●	**Improper Input Validation Out-of-bounds Write**
potrace	CVE-2013-7437	heap-buffer-overflow	v3:Medium	Numeric Errors	●	●	●	**Integer Overflow- or Wraparound Out-of-bounds Read**
w3m	CVE-2016-9438	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
w3m	CVE-2016-9443	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
w3m	CVE-2016-9622	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference
w3m	CVE-2016-9631	null-pointer-dereference	v3:Medium	NULL Pointer Dereference	●	●	●	NULL Pointer Dereference

Table 4: The ● symbol denotes an exact match, the ○ symbol denotes the we were unable to evaluate a component, ↑ denotes an increase in severity by one level, and ↓ denotes a reduction in severity by one level. We highlight cases in bold text where our approach derived a more specific CWE assignment.

methodology in Section 3 to extract the causal relationships. With those relationships extracted, we evaluated how well our approach is able to identify the type of the final error (*e.g.,* heap-use-after-free) along with the location at which the violation occurred. Next, we used `BERTScore` [83] to determine the semantic similarity between our output and the consolidated diagnosis of the professional developers. `BertScore` [83] is an evaluation metric for text generation that is designed to correlate well with human judgments on similarity. In addition, we also manually verified that the explanations were similar. Overall, for 6 of the 7 bugs listed in Table 3, our approach accurately identified the fault locations. In addition, we achieved an average BERTScore of 86%. On average, our approach took 26.10 seconds (0.37 minutes) to diagnose a crash which is a fraction of the time taken to diagnose a "Very Difficult" bug. Leveraging our approach, developers could save debugging time and effort. For bug `grep.3220317a` (considered "Moderately Difficult") we were unable to reproduce the crash in the emulator due to limitations in our prototype.

In keeping with Böhme et al. [12]'s recommendation to utilize other benchmarks for empirical evaluations, we also curated a dataset with known real-world bugs found in 12 popular programs. Each bug maps to a CVE with a crashing input and vulnerable program version. To gather our dataset, we made sure that (*i*) the vulnerable program and crashing input are publicly available and we can reproduce the bug in our emulated setting, (*ii*) a known patch, along with the initial bug report submitted by the bug hunter providing details of the failure and the Common Vulnerabilities and Exposures (CVE) report with the CVSS score and CWE, are available, (*iii*) we cover a diverse set of programs and final errors, (*iv*) we ensure a diverse representation of functionalities, covering areas such as image processing, regular expression parsing, and binary parsing and (*v*) following the recommendations of Kochhar et al. [40], we sought variability in program size, measured in lines of code. We achieved a 94% success rate in reproducing real-world bugs in our emulated setting as we had to eliminate two bugs: `CVE-2016-10272` for the `libtiff` program and `CVE-2017-6850` for the `jasper` program. Table 4 displays our final dataset, including 33 bugs across the 12 programs.

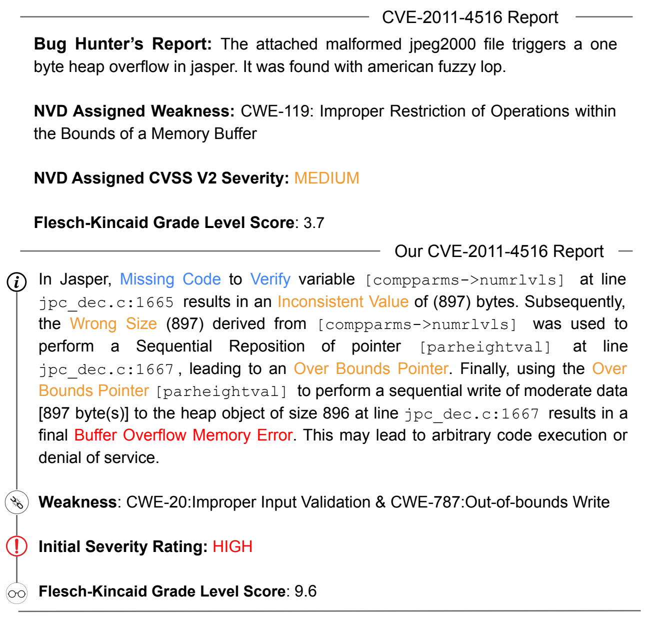
For this experiment, we manually inspected the bug reports, CVE reports, and bug-fixing commits for all bugs to extract detailed information concerning the nature of failures, their respective locations, the underlying causes of these failures, the specific locations where the causes originated, and the severity of the failure. Next, we compared how well our approach identified the type of failure (*e.g.,* use-after-free) along with the location in the source code. In all cases, our approach correctly identified the failures along with their locations. To further verify the accuracy of the 22 heap-based failures, we compared the results (*i.e.,* operation and the number of bytes read or written) of our custom sanitizer with that obtained from AddressSanitizer [58] and found that the culprit failure operations were similar.

Following a similar procedure, we compare the actual cause of the failure with our diagnosis. Specifically, we examine the first bug class in our causal relationship chain to determine the initial cause of the failure (*e.g.,* missing verification), and extract the attributes, sites (*i.e.,* locations in source code), and variables associated with the bug class. We extract that information from the submitter's report and commits as well. Our results show that we are able to identify the cause for 31 of the 33 bugs in Table 4. Our failure in the remaining cases is due to the fact that our current implementation had difficulty following macros that obscure the exact site.

Next, we compare the Common Vulnerability Scoring System (CVSS) severity levels generated by our approach for each bug with those specified in NVD. Specifically, we first focused on 26 out of the 31[2] vulnerabilities with V3 severity listed in Table 4, comparing our CVSS V3 scores with those in the NVD. Our approach arrived at the same severity level matches for 20 out of the 26 vulnerabilities. However, the scores for CVE-2015-8668, CVE-2016-7445, CVE-2017-9117, and CVE-2017-12858 were lower in our approach due to the requirement for *user interaction* through a specifically crafted file to trigger the vulnerability. In the NVD vulnerability report, however, the *user interaction* field was marked as not required, explaining the difference in scores. For the other two vulnerabilities, CVE-2017-6966 and CVE-2018-20623, our approach rated them as High severity instead of Medium. Although there is an impact on confidentiality, for some reason, that is not accounted for in those two NVD entries.

Back to the future: Due to CVSS v3 being adopted as the preferred version by the NVD after 2015, vulnerabilities published prior to then lack associated V3 scores. Nonetheless, it is important to reassess these vulnerabilities and assign V3 scores, as they are

[2]Since we were unable to diagnose the cause for 2 vulnerabilities CVE-2015-7498 and CVE-2017-9049, we did not include them in this evaluation.


CVE-2011-4516 Report

Bug Hunter's Report: The attached malformed jpeg2000 file triggers a one byte heap overflow in jasper. It was found with american fuzzy lop.

NVD Assigned Weakness: CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

NVD Assigned CVSS V2 Severity: MEDIUM

Flesch-Kincaid Grade Level Score: 3.7

Our CVE-2011-4516 Report

In Jasper, Missing Code to Verify variable `[compparms->numrlvls]` at line `jpc_dec.c:1665` results in an Inconsistent Value of (897) bytes. Subsequently, the Wrong Size (897) derived from `[compparms->numrlvls]` was used to perform a Sequential Reposition of pointer `[parheightval]` at line `jpc_dec.c:1667`, leading to an Over Bounds Pointer. Finally, using the Over Bounds Pointer `[parheightval]` to perform a sequential write of moderate data [897 byte(s)] to the heap object of size 896 at line `jpc_dec.c:1667` results in a final Buffer Overflow Memory Error. This may lead to arbitrary code execution or denial of service.

Weakness: CWE-20:Improper Input Validation & CWE-787:Out-of-bounds Write

Initial Severity Rating: HIGH

Flesch-Kincaid Grade Level Score: 9.6

Figure 4: Example of our auto-generated report with accompanying readability score, CWE labels, and initial severity rating.

occasionally still exploited in the wild. CVE-2011-0997 [7] and CVE-2004-0113 [29] serve as cases in point. For the 5 vulnerabilities in our dataset for which only V2 scores were available, we successfully produced precise V3 scores for all of them. To confirm our results, we manually analyzed each bug to determine their V3 scores.

4.2.1 *Readability and Specificity of Reports.*

For the bug reports we generate, the Flesch-Kindcaid Grade Level Score is 8.97 on average. That high score indicates that our reports contain specific information, but these details require the target audience to have a higher understanding of the subject matter. Figure 4 shows a sample report compared to that submitted by a bug finder (with a readability score of 3.7). In that example, the more specialized use of terminology related to security bugs in our report leads to the higher score of 9.6. Given we target software developers, the fact that more domain knowledge is needed to comprehend the technical details is really not an issue — as the more details provided to the developer, the easier it is to fix the bug.

As high readability is not our primary goal, we evaluate the specificity of our reports by comparing the weakness types (CWEs) we assign to the reports following the procedure outlined in Section 3.3 to the CWEs assigned by NVD or the submitter of the vulnerability report. The results in Table 4 show that for null-pointer-dereferences, heap-use-after-free, and double-frees, our CWEs are in exact alignment with the assigned CWEs. For the CWE names highlighted in bold in Table 4, we generate CWEs from the NVD slice that are more specific to the failure. Moreover, for heap-based and stack-based failures, we express vulnerabilities as a chain of weakness (*i.e.,* CWEs) linked by causality. For example, these vulnerabilities are assigned two CVEs; the first describes the initial weakness that causes the failure, and the second describes the failure itself. This refinement allows an analyst to better understand the chain of weakness types that lead to failure and devise effective mitigation strategies [53].

5 RELATED WORK

Fault localization has been an active area of research due to its importance in the bug-fixing process and the fact that it is a tedious and time-consuming task for developers and software analysts [34, 39, 63]. Fault localization, sometimes referred to as root cause analysis, is the process of finding the location of a bug based on its observed symptoms [63].

Deep learning-based fault localization. In recent years, the field of deep learning has been showing promising signs for fault localization. As such, some approaches [43, 62, 84] leverage deep neural networks for fault localization due to their ability to learn code features and semantics.

Spectrum-based Methods. A number of solutions [2, 11, 51, 56, 60, 82] leverage spectrum-based methods to perform fault localization by analyzing the deviations between a set of successful and failed test cases. Blazytko et al. [11] introduced AURORA to identify and explain the root cause of crashes generated during fuzzing. The root cause is presented as a list of the top 50 predicates that contributed to the crash. To identify the root cause, the authors first perform exploration fuzzing to derive a set of crashing and non-crashing inputs. Next, they execute the program in an instrumented environment to derive execution traces for the crashing and non-crashing inputs. Finally, they analyze the semantic divergence among the traces to pinpoint the root cause. Similarly, Shen et al. [60] use statistics to pinpoint the root cause of a crash.

Data Flow Analysis. In contrast to spectrum-based methods, other approaches [21, 48, 77, 78, 81] rely mainly on dataflow analysis to identify the root cause by performing backward dataflow analysis from the site of a crash. Cui et al. [21] proposed an approach called RETracer that identifies the root cause of crashes in production systems. They define the root cause at function-level granularity as the first function to propagate a value that causes a crash. To identify the root cause, the authors reconstruct the program data flow from a core dump beginning at the crash point and perform backward taint propagation. However, the data-flow analysis could fail if the core dump gets corrupted.

Xu et al. [79] later introduced POMP, a technique for identifying the root cause of crashes from core dumps with or without corrupted data. Specifically, the authors augment the core dump with a hardware-based instruction trace which they collect after a crash. Xu et al. [79] uses both artifacts to reconstruct the program data flow that led to the crash. In this context, the root cause is given as the minimum assembly-level instructions contributing to the crash. The authors later proposed extensions [48, 49]) to improve the data-flow analysis process. Similarly, Cui et al. [20] proposed REPT, which combines hardware-based instruction traces and a core dump to reconstruct the program data flow. In contrast to Xu et al. [79], their objective is to facilitate reverse debugging of software failures in deployed systems. Unfortunately, in the presence of instructions that operate in irreversible ways, systems like theirs that do not track concrete values of registers or memory at runtime can only approximate the data flow leading to the crash [21, 48, 77, 78]. Our approach tracks enough information during execution to determine definitive connections between instructions.

More recently, Yagemann et al. [81] introduced ARCUS and extensions [80], which combine hardware-based instruction traces and symbolic execution to identify the root cause of exploits in production systems. The authors used a custom kernel integrated with hardware-based instruction tracking capabilities to record execution and track the state of memory until a violation is detected. Subsequent to a violation, the authors perform root cause analysis using symbolic execution and bug class-specific heuristics for identifying memory corruption vulnerabilities (*e.g.,* stack-overflow, heap-overflow).

While sharing the goal of identifying and explaining the cause of a crash, CrashTalk differs by design and its underlying objective. As shown later on, our approach does not impose requirements such as hardware-based tracing, a set of successful and failed test cases, or custom modifications to the kernel. In addition, a common theme among these works is that they define and present a report of the root cause from their perspective, which may not align well with the developer's needs. They also do not pay attention (like we do) to the usability of the information they output (i.e., with respect to readability and specificity). We focus on localizing and generating a complete bug report for the developer by leveraging a taxonomic model that precisely and accurately describes a bug. Furthermore, we also provide an assessment of the severity of the failure to aid in prioritizing the patch.

Lastly, bug severity assessment is an active area of research [4, 32, 33, 36, 54, 65, 68] with most approaches leveraging Machine Learning or Natural Language processing to predict different aspects of a bug's severity. However, in contrast, we do not rely on the availability of textual descriptions from vulnerability databases or the availability of labeled bug datasets to train models. Instead, we collect artifacts and use the Bugs Framework to explain the reasons for the crash and to provide severity metrics at the time of diagnosis. In that regard, our output could serve as training data for approaches that leverage Machine Learning models.

6 LIMITATIONS

Bug fixing is crucial for effective software maintenance, but the process can consume a large amount of software developers' time. To help lessen the burden on developers, numerous studies [3, 5, 6, 17, 23, 25, 42, 50, 59, 66, 73] have studied the relationship between bug reports and how long it takes to fix the reported flaws. Their findings suggest that bug reports can sometimes contain valuable information that helps project managers decide who might be the appropriate developer to address the reported problem, how to prioritize the fixes, and how to go about correcting the bug itself.

That said, these seemingly straightforward tasks can be complicated by several factors, including determining whether the bug has been previously submitted [41], determining whether it is a genuine bug [55] that can be reproduced, and assessing the severity of the bug [67]. Hence, we acknowledge that the inclusion of descriptive elements [64] like ours that help provide valuable context [61, 70] to the bug fixer is only one factor that can impact bug fixing time.

Nevertheless, solutions like ours address a much-needed capability [40, 57, 75].

Lastly, our approach aims to diagnose common memory-related flaws in C/C++ programs, such as NULL pointer dereferences, double-frees, use-after-frees, heap-based buffer overflows, and stack-based buffer overflows. As such, our datasets were limited to these types of failures. Our operational setting assumes the presence of source code and debugging information for each program to enable the generation of comprehensive bug reports that pinpoint the cause and consequences of a failure. We believe these are reasonable constraints for software developers or security analysts looking to understand the cause of a failure and create a patch from the details we provide.

7 CONCLUSION

We present an end-to-end solution for diagnosing and explaining the cause of a failure. We show that our output is similar to bug descriptions from professional software engineers by evaluating our approach on a benchmark intended for realistic evaluation of automated debugging tools. We also demonstrate the effectiveness of our approach by evaluating it on a different set of 33 real-world bugs that we painstakingly curated. Our results show that we are able to automatically diagnose over 94% of the vulnerabilities with precision matching that of human experts. To assist with bug prioritization, we also provide initial ratings of bug severity that are on par with the manually assigned ratings listed in the National Vulnerability Database.

8 AVAILABILITY

To promote further enhancements in this important area of secure software development, our data and code are available at https://github.com/kedjames/CrashTalk.

9 ACKNOWLEDGMENTS

The authors would like to express their gratitude to Irena Bojanova at NIST's Software and Systems Division for her assistance in helping us better understand the intricacies of the Bugs Framework and for her insightful comments on an earlier draft of this paper. We also thank the anonymous reviewers for their suggestions for improving our work.

REFERENCES

[1] 2020. Zeropoint Emulated Lightweight Operating System. https://github.com/zeropointdynamics/zelos.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *International Conference on Automated Software Engineering*. 88–99.

[3] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is it a bug or an enhancement? A text-based approach to classify change requests. In *Conference of the center for advanced studies on collaborative research: meeting of minds*. 304–318.

[4] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. 2021. Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021).

[5] K. Arjun. 2012. Key factors impacting on response time of software vendors in releasing patches for software vulnerabilities.

[6] Ashish Arora, Chris Forman, Anand Nandkumar, and Rahul Telang. 2010. Competition and patching of security vulnerabilities: An empirical analysis. *Inf. Econ. Policy* 22 (2010).

[7] Avaya. 2019. H.323.Deskphone and IP Conference Phone DHCP security update (CVE-2011-0997 and CVE-2009-0692).

[8] Fabrice Bellard. 2019. QEMU Tiny Code Generator. https://github.com/qemu/qemu/tree/v4.2.0/tcg

[9] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the International Symposium on Foundations of Software Engineering*.

[10] Paul Black. 2018. Juliet 1.3 Test Suite: Changes From 1.2. https://doi.org/10.6028/NIST.TN.1995

[11] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. {AURORA}: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security Symposium*.

[12] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Foundations of Software Engineering*.

[13] Irena Bojanova. 2021. The Bugs Framework (BF). https://usnistgov.github.io/BF

[14] Irena Bojanova, Paul E Black, Yaacov Yesha, and Yan Wu. 2021. The bugs framework (BF): A Structured Approach to Express Bugs. In *IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 175–182.

[15] Irena Bojanova and Carlos Eduardo Galhardo. 2021. Classifying Memory Bugs Using Bugs Framework Approach. In *IEEE Annual Computers, Software, and Applications Conference*. 1157–1164.

[16] Irena Bojanova, Carlos Eduardo Galhardo, and Sara Moshtari. 2021. Input/output check bugs taxonomy: Injection errors in spotlight. In *IEEE International Symposium on Software Reliability Engineering Workshops*.

[17] Thiago Figueiredo Costa and Mateus Tymburibá. 2022. Challenges on prioritizing software patching. *International Conference on Security of Information and Networks* (2022), 1–8.

[18] Roland Croft, M Ali Babar, and Li Li. 2022. An investigation into inconsistency of software vulnerability severity across data sources. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 338–348.

[19] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2022. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1044–1063.

[20] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *USENIX Symposium on Operating Systems Design and Implementation*.

[21] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. 2016. Retracer: Triaging Crashes by Reverse Execution From Partial Memory Dumps. In *International Conference on Software Engineering*.

[22] Aurelien Delaitre, Bertrand Stivalet, Paul Black, Vadim Okun, Terry Cohen, and Athos Ribeiro. 2018. SATE V Report: Ten Years of Static Analysis Tool Expositions.

[23] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and Muhammad Ali Babar. 2020. Software Security Patch Management - A Systematic Literature Review of Challenges, Approaches, Tools and Practices. *Information Software Technology* (2020).

[24] Marc Eisenstadt. 1997. My hairiest bug war stories. *Commun. ACM* (1997), 30–37.

[25] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. 2019. Hey Google, What Exactly Do Your Security Patches Tell Us? A Large-Scale Empirical Study on Android Patched Vulnerabilities. *ArXiv* (2019).

[26] Ed FIRST. 2015. Common Vulnerability Scoring System v3. 0: Specification Document.

[27] Ed FIRST. 2015. Common Vulnerability Scoring System v3.1: User Guide. https://www.first.org/cvss/v3.1/user-guide

[28] Jonathan Foote. 2015. Exploitable crash analyzer version 1.6. https://github.com/jfoote/exploitable.

[29] Recorded Future. 2018. Threat Actors Remember the Vulnerabilities We Forget.

[30] Carlos Cardoso Galhardo, Peter Mell, Irena Bojanova, and Assane Gueye. 2020. Measurements of the most significant software security weaknesses. In *Annual Computer Security Applications Conference*. 154–164.

[31] Christoph Gentsch, Rohan Krishnamurthy, and Thomas S. Heinze. 2021. Benchmarking Open-Source Static Analyzers for Security Testing for C. In *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*.

[32] Xi Gong, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Zhuobing Han. 2019. Joint prediction of multiple vulnerability characteristics through multi-task learning. In *International Conference on Engineering of Complex Computer Systems*. 31–40.

[33] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to predict severity of software vulnerability using only vulnerability description. In *IEEE International conference on software maintenance and evolution*.

[34] Thomas Hirsch and Birgit Hofer. 2021. What we can learn from how programmers debug their code. In *IEEE/ACM International Workshop on Software Engineering Research and Industrial Practice*.

[35] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving Bug Triage With Bug Tossing Graphs. In *ACM Foundations of Software Engineering*. 111–120.

[36] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. 2022. Evocatio:

Conjuring bug capabilities from a single poc. In *ACM Conference on Computer and Communications Security*. 1599–1613.

[37] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM Conference on Virtual Execution Environments*. 121–132.

[38] J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. 1975. *Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel*. Technical Report. Naval Research Branch.

[39] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security*.

[40] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[41] Alina Lazar, Sarah Ritchey, and Bonita Sharif. 2014. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *IEEE Working Conference on Mining Software Repositories*.

[42] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Conference on Computer and Communications Security*.

[43] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *International Conference on Software Engineering*. 661–673.

[44] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *ACM International Symposium on Software Testing and Analysis*.

[45] Douglas R McCallum and James L Peterson. 1982. Computer-based readability indexes. In *Proceedings of the ACM'82 Conference*.

[46] MITRE. 1999. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/

[47] Mitre. 2000. CWE Mapping Guide. https://cwe.mitre.org/documents/cwe_usage/quick_tips.html

[48] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. 2019. POMP++: Facilitating Postmortem Program Diagnosis with Value-set Analysis. *IEEE Transactions on Software Engineering* (2019).

[49] Dongliang Mu, Wenbo Guo, Alejandro Cuevas, Yueqi Chen, Jinxuan Gai, Xinyu Xing, Bing Mao, and Chengyu Song. 2019. RENN: Efficient reverse execution with neural-network-assisted alias analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 924–935.

[50] Vidya Murthy. 2020. Analysis: Assessing Correlation between CVSS Scores in Vulnerability Disclosures and Patching. *Biomedical instrumentation & technology* (2020), 44–46.

[51] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *Transactions on software engineering and methodology* (2011), 1–32.

[52] NIST. 2000. NVD CWE Slice. https://nvd.nist.gov/vuln/categories

[53] NIST. 2023. CVEs and the NVD Process. https://nvd.nist.gov/general/cve-process

[54] Saahil Ognawala, Ricardo Nales Amato, Alexander Pretschner, and Pooja Kulkarni. 2018. Automatically assessing vulnerabilities discovered by compositional analysis. In *International Workshop on Machine Learning and Software Engineering in Symbiosis*. 16–25.

[55] Quentin Perez, Pierre-Antoine Jean, Christelle Urtado, and Sylvain Vauttier. 2021. Bug or not bug? That is the question. *International Conference on Program Comprehension* (2021), 47–58.

[56] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*. 30–39.

[57] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. 2016. What Makes a Satisficing Bug Report? *International Conference on Software Quality, Reliability and Security* (2016), 164–174.

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

[59] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. *International Conference on Software Engineering* (2012).

[60] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *ACM Asia Conference on Computer and Communications Security*. 13 pages.

[61] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Transactions on Software Engineering* 45, 9 (2019).

[62] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *International Symposium on Software Testing and Analysis*. 273–283.

[63] Marc Solé, Victor Muntés-Mulero, Annie Ibrahim Rana, and Giovani Estrada. 2017. Survey on models and techniques for root-cause analysis. *arXiv preprint arXiv:1701.08546* (2017).

[64] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. 2020. The significance of bug report elements. *Empirical Software Engineering* (2020), 5255–5294.

[65] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. 2017. Assessment of vulnerability severity using text mining. In *Pan-Hellenic conference on informatics*. 1–6.

[66] Yida Tao, DongGyun Han, and Sunghun Kim. 2014. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. *IEEE International Conference on Software Maintenance and Evolution* (2014), 271–280.

[67] Yuan Tian, D. Lo, Xin Xia, and Chengnian Sun. 2014. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering* (2014), 1354–1383.

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

[69] Jamal Uddin, R. Ghazali, M. M. Deris, Rashid Naseem, and Habib Shah. 2016. A survey on bug prioritization. *Artificial Intelligence Review* (2016), 145–180.

[70] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. , 38-49 pages.

[71] Renan Vieira, César Lincoln Cavalcante Mattos, Lincoln S. Rocha, João Paulo P. Gomes, and Matheus Henrique Esteves Paixão. 2022. The role of bug report evolution in reliable fixing estimation. *Empirical Software Engineering* 27 (2022).

[72] Renan Vieira, Diego Mesquita, César Lincoln Mattos, Ricardo Britto, Lincoln Rocha, and João Gomes. 2022. Bayesian Analysis of Bug-Fixing Time Using Report Data. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 57–68.

[73] Daniel Votipka, Rock Stevens, Elissa M. Redmiles, Jeremy Hu, and Michelle L. Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. *IEEE Symposium on Security & Privacy* (2018), 374–391.

[74] Andreas Wagner and Johannes Sametinger. 2014. Using the juliet test suite to compare static security scanners. In *International Conference on Security and Cryptography*.

[75] E. Winter, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, V. Nowack, and J. Woodward. 2023. How do Developers Really Feel About Bug Fixing? Directions for Automatic Program Repair. *IEEE Transactions on Software Engineering* (2023), 1823–1841.

[76] Yan Wu, Irena Bojanova, and Yaacov Yesha. 2015. They know your weaknesses–do you?: Reintroducing common weakness enumeration. *CrossTalk* 45 (2015).

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

[78] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. POMP: Postmortem Program Analysis with Hardware-enhanced Post-crash Artifacts. In *USENIX Security Symposium* (Vancouver, BC, Canada). 17–32.

[79] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security Symposium*. 17–32.

[80] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *ACM Conference on Computer and Communications Security*. 17 pages.

[81] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *USENIX Security Symposium*.

[82] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. *ACM Software Engineering Notes* (2002), 1–10.

[83] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[84] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *International Conference on Software Analysis, Evolution and Reengineering*.

[85] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* (2010), 618–643.