Path-Based Fault Correlations

Wei Le and Mary Lou Soffa
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
{weile, soffa}@virginia.edu

ABSTRACT

Although a number of automatic tools have been developed to detect faults, much of the diagnosis is still being done manually. To help with the diagnostic tasks, we formally introduce *fault correlation*, a causal relationship between faults. We statically determine correlations based on the expected dynamic behavior of a fault. If the occurrence of one fault causes another fault to occur, we say they are correlated. With the identification of the correlated faults, we can better understand fault behaviors and the risks of faults. If one fault is uniquely correlated with another, we know fixing the first fault will fix the other. Correlated faults can be grouped, enabling prioritization of diagnoses of the fault groups. In this paper, we develop an interprocedural, path-sensitive, and scalable algorithm to automatically compute correlated faults in a program. In our approach, we first statically detect faults and determine their error states. By propagating the effects of the error state along a path, we detect the correlation of pairs of faults. We automatically construct a correlation graph which shows how correlations occur among multiple faults and along different paths. Guided by a correlation graph, we can reduce the number of faults required for diagnosis to find root causes. We implemented our correlation algorithm and found through experimentation that faults involved in the correlations can be of different types and located in different procedures. Using correlation information, we are able to automate diagnostic tasks that previously had to be done manually.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics, Symbolic execution*

General Terms

Algorithms, Experimentation, Reliability, Security

Keywords

fault correlation, error state, demand-driven, path-sensitive

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

1. INTRODUCTION

Today the software industry relies more and more on automatic tools to detect faults [5, 9, 12, 15]; however, many of the diagnostic tasks are still done manually, which is time-consuming. For large newly developed software, thousands or even millions of fault reports are generated by tools [5, 12, 28]; however the generated reports are processed at a very low speed, estimated as 15–20 reports per person day [1] or 120 lines of code per person hour on average [17]. Fault diagnosis, done statically on the program source code, aims to identify and fix the causes of detected faults. Diagnosing faults is challenging for a number of reasons. One reason is that the root cause can be located far from where the fault is detected, while the code around the fault can be complex. Unlike debugging, in fault diagnosis, there is no runtime information available to assist in explaining faults. Also, in static analysis, real faults are often mixed with an overwhelming number of false alarms and benign errors.

In this paper, we explore relationships among faults for fault diagnosis. We show that a causal relationship can exist between faults; that is, the occurrence of one fault can cause another fault to occur, which we call *correlation*. As an example, in Figure 1 we show a fault correlation discovered in `ffmpeg-0.4.8`. The correlation exists between an integer signedness error at node 2 and a null-pointer dereference at node 5, as any input that leads to the integer violation at node 2 triggers the null-pointer dereference at node 5 along path $\langle 1, 2, 5\rangle$. The trigger can occur because the variable `current_track` at node 2 is not guaranteed to get the unsigned value of `AV_RL32(&head[i+8])` (see the macro definition at the bottom of the figure). If a large value is assigned, the signed integer `current_track` would get a negative value at runtime. When `current_track` is negative, the branch $\langle 2, 5\rangle$ is taken and the memory allocation at node 4 is skipped, causing the dereference of `fourxm->tracks` at node 5 to encounter a null-pointer.

Fault correlation is a relationship defined on the dynamic behavior of faults. When a program runs, an initial root cause can propagate and cause a sequence of property violations along the execution before an observed symptom, e.g., crash, is detected. In traditional static tools, the dependencies of those property violations are not identified; either only the first violation is reported or all the violations are reported but as separate faults [4, 11, 22, 28]. For the above example, static detection only reports that node 2 contains an integer violation, but it cannot explain whether it is benign or malignant, and if harmful, how severe is the consequence. A static detector for null-pointer dereference also cannot discover the vulnerability, because the detector may not be aware of any integer violations. When the impact of integer fault is not considered, the static analysis would report the path $\langle 1, 2, 5\rangle$ infeasible,

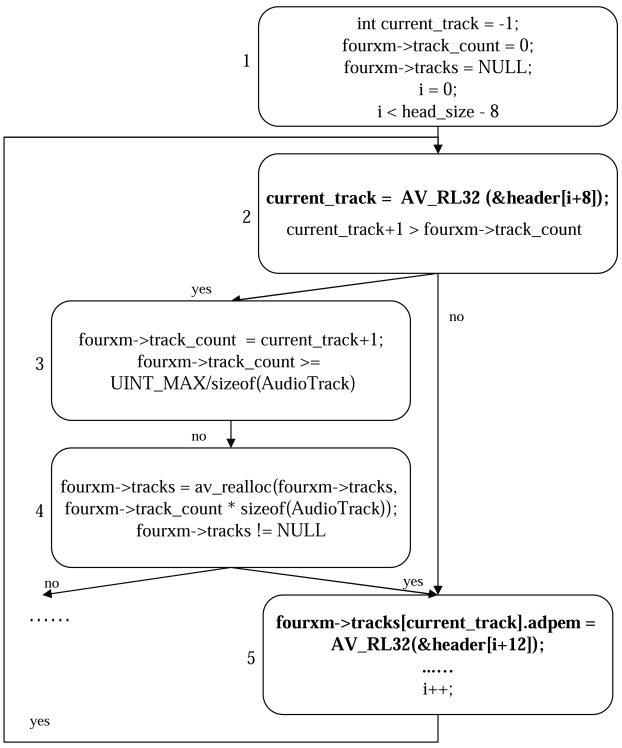

Figure 1: Fault Correlation in ffmpeg-0.4.8

as `AV_RL32(x)` always returns a non-negative integer and thus the result of the addition at node 2 should be always larger than `fourxm->track_count`'s initial value 0. However, given the fault correlation, we know that: there exists a null-pointer dereference at node 5; its root cause is the integer fault at node 2; and by fixing the integer fault, the null-pointer dereference can also be fixed.

Fault correlation helps fault management in the following ways: 1) we can detect new faults with introduced fault impact, e.g., the null-pointer dereference shown in Figure 1. These faults are impossible to be identified using traditional static detectors; 2) we can confirm and prioritize real faults by revealing their potential consequences; and 3) we can group faults based on their causes.

Determining fault correlations in current static tools is challenging for three reasons. First, identification of correlations of faults requires knowledge of fault propagation, which only can be obtained when program paths are considered; however, exhaustive static analysis based on full path exploration is not scalable. Another reason is that most static tools only detect one type of fault, while correlations often occur among faults of different types as shown in the above example. Also, in order to statically compute the propagation of a fault, the potential dynamic impact of a fault needs to be modeled, which is typically not done in the static tools.

In this paper, we formally define fault correlation and develop algorithms to determine correlations in a program. Our approach is to first statically detect faults and identify the error state that a fault potentially produces. We then statically propagate the potential dynamic impact of a fault along program paths. During propagation, either a previously detected fault can be identified as being correlated or with the introduced error state, another fault missed in the static detection can be discovered as part of correlation. We apply a demand-driven analysis to address the scalability of the path-sensitive analysis. The identified fault correlations are represented in a *correlation graph*.

We implemented our algorithm using an interprocedural, path-sensitive, demand-driven static analysis framework. We selected three types of faults, buffer overflow, integer error and null-pointer dereference, as case studies. In our experiments, we show that correlations commonly exist in real-world applications, and our algorithm can automatically compute them. Using the correlation information represented in the correlation graph, we are able to understand the impact of faults, prioritize diagnostic tasks, and group faults to speed up diagnosis.

To the best of our knowledge, our work is the first that formally defines and automatically computes fault correlations. The contributions of the paper include:

- the definition and classification of fault correlations,
- the identification of the usefulness of correlations in fault diagnosis,
- algorithms for automatically computing correlations,
- correlation graphs that integrate fault correlations on different paths and among multiple faults, and
- experiments that demonstrate the common existence of fault correlations and the value of identifying them.

In Section 2, we define fault correlation. In Section 3, we explain how to compute correlations. Section 4 presents the correlation graph. Section 5 gives experimental results. The related work is compared in Section 6, followed by the conclusions in Section 7.

2. FAULT CORRELATION

We first define program faults and fault correlation. We also provide examples to demonstrate correlations.

2.1 Program Faults

Definition 1: A *program fault* is an abnormal condition caused by the violation of a required property at a program point.

The property can be specified as a set of constraints to which a program has to conform. We focus on the faults caused by property violations, where the violation can be observed at certain program points. Those program points can be identified using code signatures. To determine a fault, we require finding at least one execution path that leads to the violation of constraints at the program point of interest.

For example, buffer overflow can occur at any buffer access if the length of the string is larger than the buffer size. An integer overflow occurs at integer arithmetic when the outcome of the arithmetic is larger than the maximum value the destination integer can store. For a data race to occur, the value of a shared variable at its use must be inconsistent under different interleaving of threads. Similarly, memory leaks occur if the allocated memory is never released when the pointers are no longer alive or reassigned to another part of memory. On the other hand, we do not consider a missing statement or a misused variable as a fault in this paper, as no property constraints are violated at a certain program point.

Definition 2: The *error state* of a fault is the set of data produced at runtime as a result of property violations.

Intuitively, an error state is the manifestation of a fault. That is, after executing a program statement, there exists a set of values

from which we can determine that property constraints are violated and a fault occurs. The set of values constitute an error state. If a crash would occur, we consider the values that cause the crash as the error state. We model the error state of a fault based on the fault type using constraints. The modeling is empirical and based on the common symptoms of faults a code inspector might use to manually determine fault propagation.

Table 1: Error State of Common Faults

Fault Type	Code Signature	Error State
buffer overflow	`strcpy(a,b)`	**len(a)**>size(a)
integer overflow	`unsigned i=a+b`	**value(i)**==value(a)+ value(b)-C
integer signedness	`int j...unsigned i=j`	**value(i)**>2^{31}-1
	`unsigned i...int j=i`	**value(j)**< 0
integer truncation	`unsigned i...uchar j=i`	**value(j)**<value(i)
resource leak	`Socket s=accept(); s=accept()`	**avail(Socket)**== avail(Socket)-1

Table 1 lists the error state for several common faults. Under *Code Signature*, we give example statements where a certain type of fault potentially occurs. Under *Error State*, we show constraints about corrupted data at the fault. The type of corrupted data is listed in bold. The first row of the table indicates that when a buffer overflow occurs, the length of the string in the buffer, `len(a)`, is always larger than the buffer size, `size(a)`. From the second to fourth rows, we simulate the effect of integer faults. When an integer overflow occurs, the value stored in the destination integer, `value(i)`, should equal the result of integer arithmetic, `value(a)+value(b)`, minus a type-dependent constant `C`, e.g., 2^{32}. Similarly, when an integer signedness error occurs, we would get an unexpected integer value. For example, when a signed integer casts to unsigned, any results larger than $2^{31}-1$ (the maximum value a signed 32 bit integer possibly stores) indicates the violation of integer safety constraints [4]. When an integer truncation occurs, for instance, between `uchar` and `unsigned` as shown in the table, the destination integer would get a smaller value than the source integer. In the last row, we use a socket as an example to show that when resource leaks occur, the amount of available resources in the system is reduced, and we model the error state as [`avail(Socket)==avail(Socket)-1`].

2.2 Correlation Definition

Suppose f_1 and f_2 are two program faults.

Definition 3: f_1 and f_2 are *correlated* if the occurrence of f_2 along path p is dependent on the error state of f_1. We denote the correlation as $f_1 \rightarrow f_2$. If f_2 only occurs with f_1 along path p, we say f_1 *uniquely correlates* with f_2, denoted as $f_1 \xrightarrow{u} f_2$.

The occurrence of f_2 along p is determined by the property constraints on a set of variables collected along p. If such variables are control or data dependent [29] on the corrupted data at the error state of f_1, f_1 and f_2 are correlated. Intuitively, given $f_1 \rightarrow f_2$, f_1 occurs first on the path, and the error state produced at f_1 propagates along p and leads to the property violation at f_2. Therefore, f_1 and f_2 have a causal relationship. Given $f_1 \xrightarrow{u} f_2$, f_1 is a necessary cause of f_2, which means, if f_1 does not occur, f_2 cannot occur. If the correlation is not unique, there is other cause(s) that can lead to f_2.

Consider Figure 2(a) in which the variable `input` stores a string from the untrusted user. A correlation exists between the buffer overflow at line 2 and the one at line 3, as there exists a valueflow on variable `a`, shown in the figure, that propagates the error state of the overflow at line 2 to line 3. When the first buffer overflow occurs, the second also occurs. The faults are uniquely correlated.

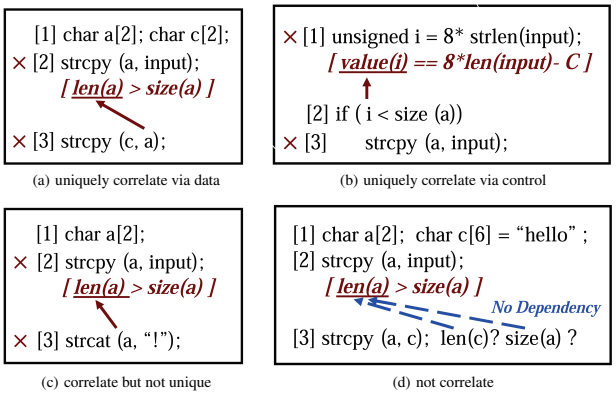

Figure 2: Defining Fault Correlation: correlated faults are marked with ×, error state is included in [ ], and corrupted data are underlined

In Figure 2(b), we show a correlation based on control dependency between faults. The integer overflow at line 1 leads to the buffer overflow at line 3, as the corrupted data, `value(i)`, produced at the integer fault impacts the conditional branch at line 2 (on which line 3 is control-dependent).

In Figure 2(c), buffer overflow at line 2 correlates with the one at line 3. However, the first overflow is not the only cause for the second because when the overflow at line 2 does not occur, the overflow at line 3 still can occur.

As a comparison, the two buffer overflows presented in Figure 2(d) are not correlated. At line 3, both the size of the buffer and the length of the string used to determine the overflow are not dependent on the corrupted data `len(a)` in the error state at line 2.

By identifying fault correlation, we can better understand the propagation of the faults and thus fault behavior. We demonstrate the value of fault correlations in two real-world programs. In the first example, we show given $f_1 \rightarrow f_2$, we can predict the consequence of f_1 through f_2, and prioritize the faults. The correlation also helps group and order faults, as in the case of $f_1 \xrightarrow{u} f_2$, fixing f_1 will fix f_2. See Example 2.

Example 1: Figure 3 presents a correlation found in the program `acpid-1.0.8`. In this example, we show how a fault of resource leak can cause an infinite loop and lead to the denial of service. The code implements a daemon that waits for connection from clients and then processes events sent via connected sockets. In Figure 3, the `while` loop at node 1 can only exit at node 5, when an event is detected by the `poll()` function at node 2 and processed by the server. Correspondingly, along the paths $\langle (1-4)^*, 1-2, 5\rangle$, the socket `fd` is created by the function `ud_accept` at node 3, and released by `clean_exit` at node 5. However, if a user does not send legitimate requests, the branch $\langle 2, 3\rangle$ is always taken, and the created sockets at node 3 cannot be released. Eventually, the list of sockets in the system is completely consumed and no socket is able to be returned from `ud_accept` at node 3. As a result, the condition `fd<0` always returns true. The execution enters an infinite loop $\langle (1-3)^*\rangle$. In this example, the impact of the resource leak makes the execution always follow the false branch of node 2 and the true branch of node 3, causing the program to hang. With fault correlation information, we can automatically identify that the root cause of the infinite loop is the resource leak. To correct this infinite loop, we can add resource release code in the loop, as shown in the figure.

Example 2: Static tools potentially report many warnings for

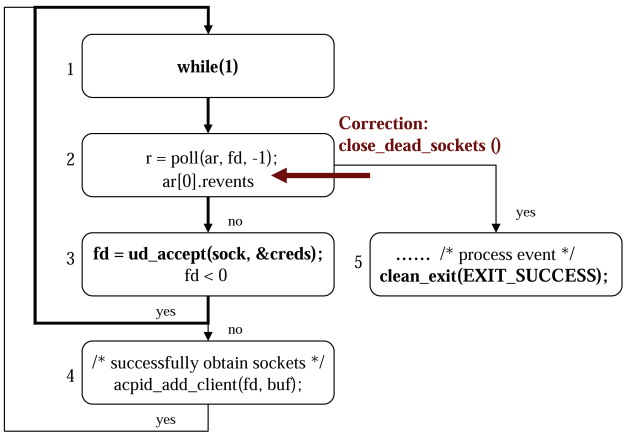

Figure 3: Correlation of Resource Leak and Infinite Loop in acpid

a program, especially when they analyze newly written code or legacy but low quality code. Consider the example in Figure 4 from `polymorph-0.4.0`. There exist 7 buffer overflows in the code, located at lines 2, 10, 12, 14, 16, 19 and 21. Although these overflows are not all located in the same procedure and even the buffers involved in the overflow are not all the same, we find that correlations exist among them. For example, the overflow at line 2 correlates with the one at line 16 along path $\langle 1-7, 16\rangle$, and line 16 correlates with line 21 along $\langle 16, 17, 21\rangle$. We can group these correlated faults and diagnose them together.

```
1  char filename[2048];
2  strcpy(filename, FileData.cFileName);
3  convert_fileName(filename);
4
5  void convert_filename(char* original){
6    char newname[2048]; char *bslash = NULL; ...
7    if(does_nameHaveUppers(original)){
8      for(i=0; i<strlen(original); i++){
9        if(isupper(original[i]))
10         { newname[i] = tolower(original[i]);
11           continue; }
12       newname[i] = original[i];
13     }
14     newname[i] = '\0';
15   }
16   else  strcpy(newname, original);
17   if(clean){
18     bslash = strrchr(newname, '\\' );
19     if(bslash != NULL) strcpy(newname, &bslash[1]);
20   } ...
21   strcpy(original, newname);
22 }
```

Figure 4: Correlations of Multiple Buffer Overflows in polymorph

To further understand the correlations in real-world programs, we conducted a study on 300 vulnerabilities in the Common Vulnerabilities and Exposure (CVE) database [8], dated between 2006-2009. We manually identified fault correlations on 8 types of common faults, including integer faults, buffer bounds errors, dereference of null-pointers, incorrect free of heap pointers, any types of resource leak, infinite loops, race conditions and privilege elevations. Our study shows that correlations commonly exist in real-world programs. In fact, the reports suggest that security experts manually correlate faults in order to understand the vulnerabilities or exploits.

Table 2 classifies the correlations we found. We mark $*$ if the fault listed in the row uniquely correlates with the fault in the column, and $\times$ for correlations that are not unique. Comparing the rows of *int* and *race* in the table, we found that integer faults and data race behave alike in correlations. Intuitively, both integer violation and data race can produce unexpected values for certain variables, and thereby trigger other faults. From the study, we also found that a fault can trigger different types of faults along different execution paths and produce different symptoms. We mark ✓ in the table if the faults from the column and row can be triggered by the same fault along different paths.

Table 2: Types of Correlated Faults Discovered in CVE

	int	buf	nullptr	free	leak	loop	race	privilege
int	$*$	$*\times$	$*$	$*$	$*$	$*\times$✓		$*$
buf	$*$	$*$	✓	$*$		✓		$*$
nullptr		✓		✓		✓		$*$
free		$*$	✓					$*$
leak			$*$			$*$		
loop	✓	$*\times$✓	✓					
race	$*$	$*\times$	$*$	$*$	$*$	$*$		$*$
privilege		$\times$						

3. COMPUTING FAULT CORRELATION

In this section, we present an algorithm to statically compute fault correlation. The approach has two phases: fault detection and fault correlation. In fault detection, we report path segments where faults occur. In fault correlation, we model the error state of detected faults and symbolically simulate the propagation of the error state along program paths to determine its impact on the occurrence of the other faults. The goals of the second phase are to identify 1) whether a fault is a cause of another fault detected in the first phase; and 2) whether a fault can activate faults that had not been identified in the first phase. As the determination of fault correlation requires path information, we use a demand-driven analysis, which has been shown to be scalable for a variety of applications [3, 10, 22].

3.1 An Overview

The steps for fault detection are shown on the left side of Figure 5. The demand-driven analysis first identifies program statements where the violation of property constraints can be observed, namely, *potentially faulty statements*. At those statements, the analysis constructs queries as to whether property constraints can be satisfied. Each query is propagated backwards along all reachable paths from where it is raised. Information is collected along the propagation to resolve the query. If the constraints in the query are resolved as `false`, implying a violation can occur, a fault is detected. The path segments that produce the fault are identified as faulty.

To improve the precision of the fault detection, we run an infeasible path detection using a similar query based algorithm, where the query is constructed at a conditional branch as to whether the outcome of the branch can always be true or false [3]. After the infeasible paths are identified and marked on the interprocedural control flow graph (ICFG) of the program, we run various fault detectors. In the fault detection, when the query that is being used to determine faults encounters an infeasible path, the propagation terminates.

Challenges of designing such a static fault detector include strategies for propagating queries and solutions for handling imprecise

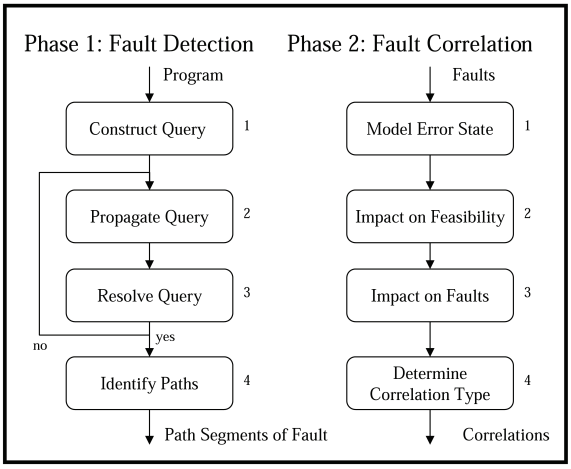

Figure 5: Fault Detection and Fault Correlation

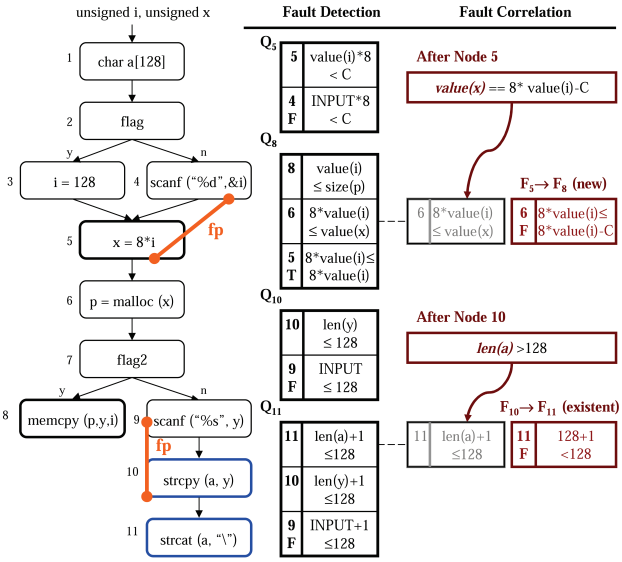

Figure 6: Correlation via Direct Impact

sources to construct and resolve symbolic integer constraints. Our query propagation is path-sensitive in that queries propagating from branches are not merged and conditional branches that are relevant to the queries are collected. The analysis is also interprocedural and context-sensitive. When a query arrives at a procedural call, we perform a linear scan of the called procedure to determine if the query can be updated. We only propagate the query into the procedure if an update is possible. We propagate the query through the loop via two iterations to determine the potential update of the query in the loop. If the loop has no impact on the query, the query advances out of the loop. If the iteration count of the loop and the update of the query in the loop can be symbolically identified, we update the query by adding the loop's effect on the original query. Otherwise, we introduce a "don't-know" tag to record the imprecision. To handle the C structure and heap, we apply an external pointer analysis, which is intraprocedural, flow-sensitive and field-sensitive. We also introduce an external constraint solver to resolve integer constraints stored in the query. More details of our analysis can be found in our technical report [23].

In the analysis, we cache queries and the resolutions at statements where the queries have been propagated. Both the cached query and the identified path segments will be reused to compute fault correlations. All the detected faults are checked for correlation in the next phase.

We developed four steps to determine the fault correlation, shown on the right in Figure 5. In the first step, we model the error state of f_1 based on its fault type (see Table 1). The error state is instrumented on ICFG as a constraint. For example, for the integer fault in Figure 1, we insert [`value(current_track)<0`] at node 2, and for the resource leak in Figure 3, we add at node 3 [`avail(Socket)==avail(Socket)-1`]. Next, we examine whether the error state of f_1 can change the results of branch correlation analysis, as an update of the conditional branch can lead to the change of feasibility, which then impacts the occurrence of f_2. In the following step, we determine the impact of f_1 directly on f_2, and finally we check if the identified correlation is unique.

3.2 Examples to Find Correlations

Based on the definition of fault correlation, for $f_1 \rightarrow f_2$ to occur, we require two conditions: 1) there exists a program path p that traverses both f_1 and f_2; and 2) along p, constraints for evaluating f_2 are dependent on the error state of f_1. In this section, we use examples to show how the steps of fault detection and fault correlation presented in Figure 5 proceed to determine the two conditions.

3.2.1 *Correlation via Direct Impact on Faults*

In Figure 6, we show an example on the left, and the actions taken in the analysis on the right. Under *Fault Detection*, we present the transitions of the query in fault detection phase. Each table describes the propagation of a query along one path. The first column of the table gives the nodes where a query propagated and updated. The second column lists the query after being updated and cached at the node. In Table Q_5, we show that, to detect integer overflow, we identify node 5 as a potentially faulty statement and raise the query [`value(i)*8<C`] (`C` is the type-dependent constant 2^{32}), inquiring whether the integer safety constraints hold. The query is propagated backwards and resolved as `false` at node 4 due to a user determined input `i`, shown in the second row of Table Q_5. Path $\langle 4, 5\rangle$ is thus determined as faulty and marked on ICFG. The query is also propagated to node 3 and resolved as `true` (this path is not listed in the figure due to space limitations). Similarly, to detect buffer overflows, we identify nodes 8, 10 and 11 as potentially faulty and raise queries to determine their safety. Table Q_8, Q_{10} and Q_{11} present the propagation of the three queries. Take Q_8 as an example. At node 8, we raise an initial query [`value(i)≤size(p)`], inquiring whether the buffer constraints are satisfied. At node 6, the query is first changed to [`8*value(i)≤value(x)`]. A symbolic substitution at node 5 further updates the query to [`8*value(i)≤8*value(i)`]. We thus resolve the query as `true` and report the buffer at node 8 safe. In the fault detection phase, we identify three faults, an integer overflow at node 5, and buffer overflows at nodes 10 and 11. We determine in the next step whether the correlation exists for these faults.

Under *Fault Correlation* in Figure 6, we list the steps for computing correlations. We first model the error state. For the integer overflow at node 5, we introduce [`value(x)==8*value(i)-C`] as an error state, shown in the first box under *Fault Correlation*. We italicized `value(x)` to indicate it is the corrupted data at this

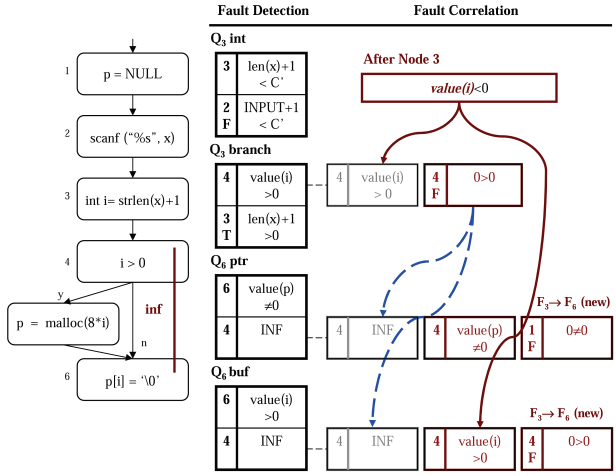

Figure 7: Correlation via Feasibility Change

fault. Conceptually, we need to propagate the error state along all program paths in a forward direction to examine if the corrupted data `value(x)` can impact the occurrence of the faults at nodes 8, 10 and 11. Since our analysis is demand-driven, to determine such impact, we actually propagate the queries raised at nodes 8, 10 and 11 in a backward direction toward the fault located at node 5, and determine if the error state can update the queries. As such backward propagation has been done in fault detection, we can take advantage of cached queries to compute correlation. In the figure, all queries listed in tables are cached in the correspondent nodes after fault detection. From Table Q_8, we discover that at the immediate successor(s) of the integer fault, i.e., node 6, query [$8*$`value(i)` $\leq$ `value(x)`] has been propagated to and is cached. The query is dependent on the corrupted data `value(x)` at the error state. We use a bold arrow in the figure to show the dependency. The query is thus updated with the error state and reaches a new resolution `false`. In this case we discover a fault that was not reported in fault detection. Using a similar approach, we introduce the error state [`len(a)>128)`] after node 10 for a buffer overflow. With this information, the query for checking buffer overflow at node 11 is resolved to `false`. In this case, two previously identified faults are determined as correlated.

To determine $f_1 \xrightarrow{u} f_2$, we examine when f_1 is fixed, whether f_2 still can occur. As for $f_1 \xrightarrow{u} f_2$, f_1 is the necessary cause of f_2, and fixing f_1 ensures the correctness of f_2. Our approach is to replace the inserted error state with the constraints that imply the correctness of the node. For example, in Figure 6, we replace the error state at node 5 with [`value(x)==8*value(i)`], and at node 10 with [`len(a)`$\leq$`128`]. With the new information, node 8 is determined as safe, indicating the correlation of node 5 and node 8 is unique, while node 11 still reports unsafe, showing the correlation between nodes 10 and 11 is not unique.

In our approach, the two conditions for determining fault correlation are ensured by two strategies. First, in fault correlation, if queries are updated with the error state of f_1 and still not resolved, we continue propagating the updated query along the faulty path of f_1, which assure f_2 and f_1 are located along the same path. For instance, in the above example, if the buffer overflow query raised at node 8 is not resolved at node 5 with the error state, it would continue to propagate along path $\langle 5, 4\rangle$ for resolution, as the error state is only produced along the faulty path $\langle 5, 4\rangle$. Second, we establish the dependency between f_2 and f_1 by assuring the error state of f_1 can update the queries of f_2 and the variables in the queries are dependent on the corrupted data in the error state.

3.2.2 *Correlation via Feasibility Change*

The error state of f_1 also can impact f_2 indirectly by changing the conditional branches f_2 depends upon, shown in Figure 7. The program is a simplified version of Figure 1. Under *Fault Detection*, we list the query transitions to detect infeasible paths and faults. Under *Fault Correlation*, we show the query update in fault correlation. In this example, our focus is to present how an integer error found at node 3 changes the branch correlation at node 4 and then impacts other faults. An error state [`value(i)<0`] is modeled after node 3. Examining cached query at node 4, we find that the error state can update the branch query [`value(i)>0`] and resolve it to `false`. The change of the resolution implies that the path this query propagated along is no longer infeasible as identified before. Therefore, all the queries that are control dependent on this branch are potentially impacted, and we need to evaluate all the queries cached at node 4 for new resolutions. For example, we restart the query [`value(p)`$\neq$`0`] from node 4 and resolve it at node 1 as `false`, and a null-pointer dereference is discovered. Similarly, we restart the buffer overflow query [`value(i)>0`] at node 4, where we find the query is resolved as `false` with the information from the error state. In this case, the error state of the integer fault first impacts the branch and activates the propagation of the query at node 4; then the error state also has a direct impact on the query and changes its resolution to `false`.

3.3 The Algorithm of Fault Correlation

For identifying fault correlations, Algorithm 1 takes the inputs $icfg$ and n, where $icfg$ represents the ICFG with fault detection results (including the cached queries and marked faulty paths), and n is the node where the fault is detected. Our goal is to identify all the correlations for the fault at node n.

At line 1, we model the error state. For each query cached at the immediate successor(s) of the fault, we identify queries that are dependent on the error state. See lines 2–4. If the query is resolved after updating with the error state, we add it to the set of resolved queries A at line 6. Otherwise, if the updated query was used to compute faults, we add it to the list FQ at line 7. If the query was used to compute branch correlation, we add it to the list IQ at line 9. Lines 10–11 collect queries stored at the branch $q'.raise$. The faults associated with these queries are potentially impacted by the feasibility change, and thus need to be reevaluated. After queries are classified to the lists FQ and IQ, we compute the feasibility change at line 16 using IQ and then determine the impact of the error state directly on the faults at line 17 using FQ.

The determination of the resolutions of updated queries is shown in `Resolve` at line 18. The analysis is backwards. At line 20, we first propagate the queries to the predecessors of the faulty node. We then use a worklist to resolve those queries at lines 22–27. `Propagate` at line 29 indicates that we need to only propagate the queries along feasible and faulty paths. After a query is resolved at line 25, we identify paths and mark them on ICFG at line 28. For branch query, they are adjusted infeasible paths, while for queries to determine faults, the paths show where the correlation occurs.

4. CORRELATION GRAPHS

Our algorithm computes the correlation between pairs of faults. We integrate individual fault correlations in a graph representation to present correlations among multiple faults and along different paths for the whole program.

```
   Input  : ICFG with fault detection results (icfg);
            faulty node (n)
   Output: Correlations for n
 1 er = ModelErrState (n);
 2 foreach m ∈ Succ(n) do
 3     foreach q ∈ Q[m] do
 4         q' = UpdateWithErrState (er, q);
 5         if q' ≠ q then
 6             if q'.an = resolved then add q' to A
 7             else if IsFaultQ(q') then add q' to FQ
 8             else
 9                 add q' to IQ
10                 foreach x ∈ Q[q'.raise] do
11                 if IsFaultQ(x) then add x to FQ
12             end
13         end
14     end
15 end
16 Resolve(IQ)
17 Resolve(FQ)
18 Procedure Resolve (querylist Q)
19 foreach q ∈ Q do
20     foreach p ∈Pred(n) do Propagate (n, p, q)
21 end
22 while worklist ≠ ∅ do
23     remove (i, q) from worklist
24     UpdateQ(i, q)
25     if q.an = resolved then add q to A
26     else foreach p ∈Pred(i) do Propagate (i, p, q)
27 end
28 IdentifyPath(A)
29 Procedure Propagate (node i, node p, query q)
30 if OnFeasiblePath(i, p, q.ipp)∧
31    OnFaultyPath(i, p, q.fpp) then
32 add (p, q) to worklist
```

Algorithm 1: Compute Fault Correlations

Definition 4: A *correlation graph* is a directed and annotated graph $G = (N, E)$, where N is a set of nodes that represent the set of faults in the program and E is a set of directed edges, each of which specifies a correlation between two faults. The *entry nodes* in the graph are nodes that do not have incoming edges, and they are the faults that occur first in the propagation. The *exit nodes* are nodes without outgoing edges, and they are the faults that no longer further propagate. Annotations for a node introduce information about a fault, including its location in the program, the type, and the corrupted program objects at the fault if any. Annotations for the edge specify whether the correlation is unique and also the paths where the correlation occurs.

The correlation graph groups faults of the related causes for the program. The entry nodes of the graph and the nodes whose correlation are not unique should be focused to find root causes. Using the correlation graph, we can reduce the number of faults that need to be inspected in order to fix all the faults. In Figure 8, we show the correlation graphs for examples we presented before, Figure 8(a) for Figure 1, 8(b) for Figure 3, and 8(c) for Figure 4.

In Figure 1, we have shown a correlation of integer fault and null-pointer dereference along path $\langle 1, 2, 5\rangle$. Actually the integer fault at node 2 also correlates with a buffer bounds error at node 5 along path $\langle (1-5)^+, 1, 2, 5\rangle$. See Figure 8(a). If the buffer bounds error continues to cause privilege elevation, the correlation graph would show a chain of correlated faults to help understand the exploitability of the code. On the other hand, if both the null-pointer dereference and buffer underflow at node 5 are reported via a dynamic

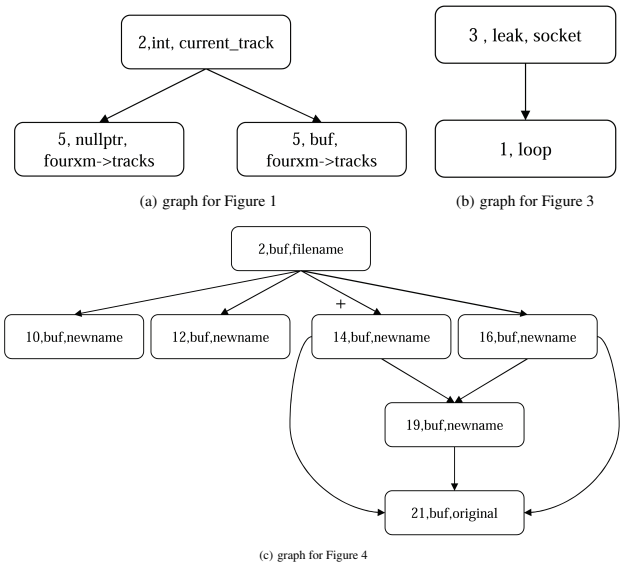

Figure 8: Correlation Graphs for Examples: + marks a correlation that is not unique

detector, using the correlation graph, we are able to know the two failures are attributable to the same root cause and can be fixed by diagnosing the integer fault at node 2. Similarly, the relationship of the resource leak and infinite loop shown in Figure 3 is depicted in Figure 8(b).

The correlation graph in Figure 8(c) integrates all correlations for 7 buffer overflows in Figure 4. To use this graph for diagnosis, we start from the entry node of the graph, as it indicates the root cause of all 7 correlated faults. Diagnosing the entry node we discover that when the input `FileData.cFileName` is copied to the `filename` buffer at line 2, no bounds checking is applied. We thus introduce a fix for line 2. The correlation graph indicates that all other correlated faults can be fixed except the fault at line 14, as in the graph, the edge from the fault at line 2 to the fault at line 14 indicates the existence of an additional root cause. We thus diagnose line 14 and introduce the second fix.

5. EXPERIMENTAL RESULTS

To demonstrate that we are able to automatically compute fault correlations and show that fault correlations are helpful for fault diagnosis, we implemented our techniques using Microsoft's Phoenix framework [26] and Disolver constraint solver [16]. We choose three types of common faults as case studies: buffer bounds error, integer truncation and signedness errors, and null-pointer dereference. In the experiments, we first run fault detection and update the ICFG with faults detected. We model the error state of integer and buffer faults using the approaches shown in Table 1 and then determine the fault correlation. It should be noted that although in our experiments, we use our fault detector to identify faults and then compute fault correlations, our technique is applicable when faults are provided by other tools. We used a set of 9 programs for experimental evaluation: the first five are selected from benchmarks that are known to contain 1–2 buffer overflows in each program [24, 33]; the rest are deployed mature applications with a limited number of faults reported by our fault detector. The experimental data about fault correlation are presented in the following four sections. The results have been confirmed by manual inspection.

Table 3: Automatic Identification of Fault Correlations

Benchmarks	Size	Faults from Detection			Fault Correlations						Faults during Correlation
		buf/corr	*int/corr*	*ptr/corr*	*int_int*	*int_buf*	*int_ptr*	*buf_buf*	*buf_int*	*total*	
wuftp-1	0.4 k	4/4	0	0	0	0	0	7	0	7	0
sendmail-6	0.4 k	0	3/1	0	0	1	0	0	0	1	1 (buf)
sendmail-2	0.7 k	4/4	0	1/0	0	0	0	3	0	3	0
polymorph-0.4.0	1.7 k	8/8	0	0	0	0	0	13	0	13	0
gzip-1.2.4	8.2 k	9/9	15/7	0	0	7	0	9	6	22	1 (buf)
ffmpeg-0.4.8	39.8 k	0	6/2	1/0	0	10	1	0	0	11	11 (1 ptr, 10 buf)
putty-0.56	66.5 k	7/6	4/2	0	3	3	0	4	0	10	5 (3 int, 2 buf)
tightvnc-1.2.2	78.9 k	0	11/8	0	9	8	0	0	0	17	7 (2 int, 5 buf)
apache-2.2.4	418.8 k	0	2/0	5/0	0	0	0	0	0	0	0

5.1 Identification of Fault Correlations

In the first experiment, we show that fault correlations can be automatically identified. Table 3 displays identified correlations. In the first two columns of the table, we list the 9 benchmark programs and their sizes in terms of lines of code. Under *Faults from Detection*, we display the number of faults identified for each program in our fault detection. Buffer bounds errors are reported in Column *buf/corr*. Integer faults are listed in Column *int/corr* and the null-pointer dereferences are shown in Column *ptr/corr*. In each column, the first number gives the identified faults and the second lists the number of detected faults that are involved in fault correlation. Our fault detector reports a total of 80 faults of three types, 51 of which are involved in fault correlation.

Under *Fault Correlations*, we list the number of pairs of faults in the program that are found to be correlated. For example, under *int_buf*, we count the pairs of correlated faults where the cause is an integer fault, which leads to a buffer overflow. Comparing the integer faults involved in the correlations under *int_buf* and *int_ptr* with the ones found in fault detection, we can prioritize the integer faults with severe symptoms. In the last column of *Fault Correlations*, we give a total number of identified correlations. In our experiments, we found fault correlations for 8 out of 9 programs. Correlations occur between two integer faults, an integer fault and a buffer overflow, an integer fault and a null-pointer dereference, two buffer overflows, as well as a buffer overflow and an integer fault.

The experiments also validate the idea that the introduction of error states can enable more faults to be discovered. We identify a total of 25 faults during fault correlation from 5 benchmarks, including buffer overflows, integer faults, and null-pointer dereferences, shown under *Faults during Correlation*.

Consider the benchmark `gzip-1.2.4` as an example. We discover a total of 25 faults and 22 pairs of them are correlated. A new buffer overflow is found after introducing the impact of an integer violation. Buffer overflow correlates with integer fault when `strlen` is called on an overflowed buffer which later is assigned to a signed integer without proper checking. We also found that the new faults generated during fault correlation can further correlate with other faults. In `putty-0.56`, two integer faults found during fault correlation resulted from another integer fault are confirmed to enable a buffer overflow. The propagation of these faults explains how the buffer overflow occurs.

5.2 Characteristics of Fault Correlations

We also collected the data about the characteristics of fault correlations, shown in Table 4. In Column *Unique/Not*, we count, for all the correlations identified, how many are uniquely correlated (see the first number in the column) and how many are not (see the second number). The data demonstrate that both types of correlations exist in the benchmarks. Column *Dir/Indir* shows whether a correlation occurs directly between two faults or indirectly as a result of feasibility change. The first number summarizes the direct correlations and the second number counts the indirect ones. The results show that most correlations are discovered via direct query interactions, and only two programs report the correlations identified from feasibility change. We also investigated the distances between the correlated faults. The experimental data under *Inter/Intra* show that along the correlated paths, the two faults can be located either intraprocedurally or interprocedurally. Therefore an interprocedural analysis is required for finding all correlations. A related metric is the distance of correlated faults along the correlation paths in terms of number of procedures. Column *Corr-Proc* gives both the minimum and maximum numbers of procedures between two correlated faults in the benchmark. We are able to find the correlation where two faults are 19 procedures apart.

Table 4: Characteristics of Fault Correlations

Benchmarks	Unique/Not	Dir/Indir	Inter/Intra	Corr-Proc
wuftp-1	4/3	7/0	7/0	1–10
sendmail-6	1/0	1/0	0/1	1–1
sendmail-2	0/3	3/0	0/3	1–1
polymorph-0.4.0	11/2	13/0	8/5	1–3
gzip-1.2.4	12/10	21/1	15/7	1–19
ffmpeg-0.4.8	11/0	1/10	0/11	1–1
putty-0.56	10/0	10/0	2/8	1–3
tightvnc-1.2.2	14/3	17/0	16/1	1–2

5.3 Computing Correlation Graphs

A correlation graph is built for each benchmark in the experiments. In Table 5, we report the total number of nodes in the correlation graph in Column *Node*. The nodes include faults identified from fault detection and fault correlation. The types of identified faults are listed in Column *Type*. For example, for the program `ffmpeg-0.4.8`, we find faults of all three types. In Column *Group*, we provide the number of groups of correlated faults for each program. We obtained the number by counting the connected components in each correlation graph. The results show that although the number of faults can be high in a program, many of the faults can be grouped and diagnosed together. For 7 out of 9 programs, the faults are clustered to less than a half of fault groups which will assist diagnosis.

Under *Analysis Cost*, we report the analysis costs for computing correlation graphs, including the time used for detecting faults (see the first number in the column) and the time used for computing fault correlations (see the second number). The machine we used to run experiments is the Dell Precision 490, one Intel Xeon 5140 2-core processor, 2.33 GHz, and 4 GB memory. The experimental data show that the analysis cost for fault detection is not always proportional to the size of the benchmarks; the complexity of the

code also matters. For example, the analysis for `sendmail-6` takes a long time to finish because all the faults are related to several nested loops. The additional costs of computing fault correlations for most of the benchmarks are under seconds or minutes, except for `gzip-1.2.4`, which contains the most faults among the benchmarks and many faults are found to impact a large chunk of the code in the program. The data suggest that the important factors that determine the analysis cost of fault correlation are the number of faults and the complexity of their interactions.

Table 5: Correlation Graphs and their Analysis Costs

Benchmarks	Size	Node	Type	Group	Analysis Cost
wuftp-1	0.4 k	4	1	1	3.9 m/43.2 s
sendmail-6	0.4 k	4	2	3	108.0 m/5.6 s
sendmail-2	0.7 k	5	2	2	10.8 s/3.7 s
polymorph-0.4.0	1.7 k	8	1	1	39.4 s/9.3 s
gzip-1.2.4	8.2 k	25	2	9	29.3 m/90.0 m
ffmpeg-0.4.8	39.8 k	18	3	7	114.2 m/3.4 m
putty-0.56	66.5 k	16	2	7	62.8 m/1.2 m
tightvnc-1.2.2	78.9 k	18	2	6	60.3 m/2.4 m
apache-2.2.4	418.9 k	7	2	7	217.8 m/2.1 s

5.4 False Positives and False Negatives

In our experiments, both false positives and false negatives have been found. Because we isolate don't-know warnings for unresolved library calls, loops and pointers, our analysis does not generate a large number of false positives. In fault correlation, we consider the following two cases as false positives: 1) at least one of the faults involved in correlation is false positive; and 2) both faults in the correlation are real faults, but they are not correlated. In our buffer overflow detection, we report a total of 7 false positives for all programs, 1 from `sendmail-6`, 4 from `gzip` and 2 from `putty`. For integer fault detection, we report a total of 10 false positives, 3 from `sendmail-6`, 2 from `polymorph`, 2 from `ffmpeg`, 1 from `putty` and 2 from `apache`. We find 25 correlations reported are actually false positives, 23 of which are related to case (1), and 2 to case (2) where the correlation paths computed are confirmed as infeasible. However, we did not find that any new faults reported during fault correlation (see the last column in Table 3) are false positives. Interestingly, we found false positive faults can correlate with each other and thus be grouped. In our implementation, we have applied such correlations to quickly remove false positives and improve the precision of our analysis. We exclude the false positives when reporting the faults and fault correlations in Tables 3, 4 and 5.

We miss fault correlations mainly in two cases: 1) we report correlated paths between two faults as don't-know; and 2) the correlation occurs among the types of faults not investigated in our experiments. For example, in the benchmark `tightvnc-1.2.2`, three integer faults are reported as not correlated, shown under *Faults from Detection* in Table 3; however, our manual inspection discovers that these faults can cause buffer read overflow, which was not considered in our fault detection.

6. RELATED WORK

Fault correlations identify a causal relationship between faults. The key idea for computing correlations is to statically simulate the propagation of a potential error state of a fault along program paths. Research in fault propagation has been done for software security [6, 13]. To understand the severity of certain types of static faults, Ghosh *et al.* injected faults in programs and dynamically triggered faults to observe their propagation and impact [13]. Chen *et al.* discovered that a successful attack performs a set of stages. The finite state machines can be used to model the activities at each stage [6]. Similar to our research, both of the above works emphasize the importance of fault propagation. However, Ghosh *et al.* obtained fault propagation by running the program, and thus the number of paths that could be explored was limited by the program inputs, while Chen *et al.* manually identified fault propagation. Fault propagation is also useful for software debugging. Using dynamic tainting, Clause *et al.* isolated the input that potentially causes failure [7]. Their interest was to find the part of program input that has dependencies with the error state.

Fault ranking aims to prioritize real and important faults for static warnings. Often, many factors can indicate the importance of a warning, such as the complexity of the code where the warning is reported or the feedback from code inspectors. Ruthruff *et al.* developed logistic regression models to coordinate those factors [27]. Kremenek *et al.* observed that warnings can be clustered in that either they were all false positives or were real faults. Thus diagnosing one can predict the importance of other faults in the cluster [20, 21]. Heckman *et al.* identified alert characteristics and applied machine learning techniques to classify actionable and non-actionable static warnings [18]. Compared to the above works which are all based on empirical observations, our approach statically groups and orders faults based on the inherent causality between faults, and thus is generally applicable.

Research in fault localization aims to automatically identify the root cause of faults. Ball *et al.* developed a localization technique for error traces generated from the model checker. The key was to identify the transitions that only appear in error traces but not correct traces [2]. There are also the approaches of delta debugging [31], dynamic value replacement [19] and coverage based fault localization [32]; however, those approaches are only applicable when the inputs that trigger the faults are available.

Research efforts have been reported on a different type of correlation in the testing area. The focus of one effort was to discover how an error can potentially mask another and impact testing coverage [30]. Another study investigated how to propagate an error to the output of the program so that its consequence can be observed [14]. Although both our research and their research explored the relationships of software defects, the faults we focused on are different from the errors studied in their work. Our work is set in the domain of statically identifiable faults, while their work focused on errors in testing.

In prior work, other types of correlations have been proposed. Mueller *et al.* developed compiler optimizations based on branch correlation [25], while Bodik *et al.* identified infeasible paths using branch correlation [3]. ESP made the assumption that there exist correlations between the outcome of branches and property states, based on which, a path-sensitive program verification can be optimized to linear complexity [9].

7. CONCLUSIONS

As faults become more complex, manually inspecting individual faults becomes ineffective. To help with diagnosis, this paper shows that identifying a causal relationship among faults helps understand fault propagation and group faults of related causes. With the domain being statically identifiable faults, this paper introduces definitions of fault correlation and correlation graphs, and presents algorithms for their computation. Our experiments demonstrate that fault correlations exist in real-world software, and we can automatically identify them. The benchmarks used in our experiments are mature applications with few faults. However, determining correlation is especially important for newly developed or developing

software which would have many more faults. Although the fault correlation algorithm is tied to our fault detection for efficiency, a slightly modified correlation algorithm would work if faults are discovered by other tools and presented to the correlation algorithm.

8. REFERENCES

[1] Personal communication with Mingdong Shang, Code Reviewer at Microsoft, 2006.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2003.

[3] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1997.

[4] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Symposium on Network and Distributed Systems Security*, 2007.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 2000.

[6] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer. A data-driven finite state machine model for analyzing security vulnerabilities. *In International Conference on Dependable Systems and Networks*, 2003.

[7] J. Clause and A. Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on software testing and analysis*, 2009.

[8] CVE. `http://cve.mitre.org/`.

[9] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[10] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 1997.

[11] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 1996.

[12] FindBugs. http://findbugs.sourceforge.net/.

[13] A. K. Ghosh, T. O'Connor, and G. Mcgraw. An automated approach for identifying potential vulnerabilities in software. In *1998 IEEE Symposium on Security and Privacy*, 1998.

[14] T. Goradia. Dynamic impact analysis: a cost-effective technique to enforce error-propagation. *SIGSOFT Software Engineering Notes*, 1993.

[15] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceeding of the 28th International Conference on Software Engineering*, 2006.

[16] Y. Hamadi. Disolver: A Distributed Constraint Solver. Technical Report MSR-TR-2003-91, Microsoft.

[17] L. Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 2008.

[18] S. Heckman and L. Williams. A model building process for identifying actionable static analysis alerts. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, 2009.

[19] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008.

[20] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. *SIGSOFT Software Engineering Notes*, 2004.

[21] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Static Analysis Symposium*, 2002.

[22] W. Le and M. L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[23] W. Le and M. L. Soffa. General, scalable path-sensitive fault detection. *Technical Report CS-2010-11 by Computer Science Department, University of Virginia*, 2010.

[24] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[25] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995.

[26] Phoenix. `http://research.microsoft.com/phoenix/`.

[27] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, 2008.

[28] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[29] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *Proceedings of the Third International Symposium on Static Analysis*, 1996.

[30] K. Wu and Y. Malaiya. The effect of correlated faults on software reliability. In *Proceedings of Software Reliability Engineering, 4th International Symposium on*, 1993.

[31] A. Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Software Engineering Notes*, 1999.

[32] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium*, 2009.

[33] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering*, 2004.