

**UNIVERSITY OF OSLO**  
**Department of Informatics**

**Program Crash  
Analysis:  
Evaluation and  
Application of  
Current Methods**

Master thesis  
60 credits

Håkon Krohn-Hansen

**26th April 2012**





# Program Crash Analysis: Evaluation and Application of Current Methods

Håkon Krohn-Hansen

26th April 2012



# Abstract

After decades of development in computer science, memory corruption bugs still pose a threat to the reliability of software. Automatic crash reporting and fuzz testing are effective ways of gathering information about program bugs. However, these methods can potentially produce thousands of crash dumps, motivating the need for grouping and prioritizing crashes. In addition, the time necessary to analyze the root cause of crashes and to implement a reliable fix in source code should be reduced.

This thesis demonstrates how fuzzing can produce a large set of different crashes in a real program. An empirical study explores methods for analyzing these crashes. Automatic bucketing and classification is performed. Call stack based grouping algorithms are compared, and modifications are suggested. Taint analysis is demonstrated as a complementary method to automatic classification based on crash dumps. Dynamic analysis using execution traces is demonstrated as a method for root cause analysis. The empirical study suggests some general results regarding program crash analysis.

Crashes should be grouped based on related crash locations and identified similarities in call stacks. A distance algorithm can be used for call stack based grouping and to identify relations between groups. It is suggested that a weighted priority model should be used for prioritizing crashes based on a strategic policy. Some possible metrics are frequency, reliability, severity estimate and relations to already fixed bugs. In order to properly fix a memory corruption bug, the underlying cause should be understood at machine-level. Execution traces with logged operands, differential debugging, Crash Graphs and input analysis might help developers analyze different aspects of memory corruption bugs.



# Contents

<b>I</b>	<b>Introduction</b>	<b>1</b>
<b>1</b>	<b>Background</b>	<b>3</b>
1.1	Bugs in commercial software . . . . .	3
1.2	Developing reliable software . . . . .	4
<b>2</b>	<b>Thesis</b>	<b>9</b>
2.1	Motivation . . . . .	9
2.2	Objective . . . . .	10
2.3	Thesis layout . . . . .	10
<b>II</b>	<b>Theory</b>	<b>11</b>
<b>3</b>	<b>Program crash analysis</b>	<b>13</b>
3.1	Memory corruption . . . . .	13
3.2	Crash dumps . . . . .	18
3.3	Crash dump analysis . . . . .	21
<b>4</b>	<b>Fuzzing</b>	<b>27</b>
4.1	Fuzzing strategies . . . . .	27
4.2	Strategies for triaging errors . . . . .	30
4.3	Assessing the effectiveness of a fuzzer . . . . .	32
<b>5</b>	<b>Dynamic program analysis</b>	<b>35</b>
5.1	Methods of dynamic analysis . . . . .	35
<b>6</b>	<b>Related research</b>	<b>39</b>
6.1	Automatic crash reporting and analysis . . . . .	39
6.2	Call stack analysis . . . . .	40
6.3	Program crash analysis using execution traces . . . . .	43
<b>III</b>	<b>Methods</b>	<b>47</b>
<b>7</b>	<b>Planning the thesis</b>	<b>49</b>
7.1	Methods overview . . . . .	49
7.2	Choosing a target program . . . . .	51
7.3	Fuzzing strategy . . . . .	53

## CONTENTS

---

7.4	Crash reliability analysis . . . . .	54
7.5	Comparison of grouping algorithms . . . . .	55
7.6	Dynamic analysis . . . . .	56
<b>IV</b>	<b>Empirical results</b>	<b>57</b>
<b>8</b>	<b>Fuzzing results</b>	<b>59</b>
8.1	Crash statistics . . . . .	59
8.2	Results of the crash verification process . . . . .	63
<b>9</b>	<b>Crash analysis</b>	<b>65</b>
9.1	Call stack analysis . . . . .	65
9.2	Prioritizing crashes . . . . .	69
9.3	Root cause analysis . . . . .	73
<b>V</b>	<b>Discussion</b>	<b>79</b>
<b>10</b>	<b>Answering questions about program crashes</b>	<b>81</b>
10.1	RQ1: How are crashes related? . . . . .	81
10.2	RQ2: How should crashes be prioritized? . . . . .	91
10.3	RQ3: How should crashes be fixed? . . . . .	94
10.4	Automatic program crash analysis . . . . .	100
<b>VI</b>	<b>Conclusion</b>	<b>101</b>
<b>11</b>	<b>Conclusion</b>	<b>103</b>
11.1	Major contributions . . . . .	103
11.2	Summary of results . . . . .	104
11.3	Critical evaluation . . . . .	106
11.4	Future work . . . . .	107
11.5	Final remarks . . . . .	108
	<b>Appendices</b>	<b>109</b>
<b>A</b>	<b>Mutation of a fuzzing template</b>	<b>111</b>
<b>B</b>	<b>!exploitable rules</b>	<b>113</b>
<b>C</b>	<b>Derivation of an expression for <math>Z(i)</math></b>	<b>115</b>
<b>D</b>	<b>List of unique crashes</b>	<b>117</b>
<b>E</b>	<b>Call stack grouping</b>	<b>119</b>



# List of Figures

1.1	Microsoft Security Development Lifecycle (SDL)	4
3.1	Paging Unit's View of 32-bit Linear Address	14
3.2	Page Translation	15
3.3	Example of a program crash	16
3.4	Example C program	19
3.5	Windows Error Report	20
3.6	Function call	21
3.7	Function prolog	22
3.8	Stack layout	22
3.9	Crash scenario 1: 16 characters - Read Access Violation	24
3.10	Crash scenario 2: 100 characters - Write Access Violation	25
3.11	Crash scenario 3: 27 characters - Unknown crash location	26
4.1	Basic blocks of the main function in <code>overflow.c</code>	29
4.2	Converging curve of unique crashes	34
5.1	Differential debugging with BinNavi	38
6.1	A two-level grouping of crashes using representative traces	41
6.2	Crash Graph	42
8.1	Progress of fuzz testing	61
8.2	Frequency count of unique crashes	62
9.1	Call stack reconstruction	66
9.2	Top-down comparison algorithm for call stack grouping	67
9.3	Crash graph of all stack frames	69
9.4	Taint information from three crashes	72
9.5	Excerpts from <code>gstype42.c</code>	77
9.6	Overflow of the buffer <code>pts</code> caused by <code>append_simple</code>	78
10.1	Expanding sort tree	82
10.2	A granular distance between individual stack frames	86
10.3	Related crashes shown in a crash graph	87
10.4	Crash graph of call stacks related by <code>gs_type1_interpret</code>	88
10.5	Automatic analysis of related crash locations	90
10.6	Triggers and symptoms of a bug	99

A.1 Mutation of a Type42 font description in a PostScript file . . .	111
--	-----

# List of Tables

4.1	Fuzzing taxonomy . . . . .	28
4.2	An example distribution of unique crashes . . . . .	33
8.1	Severity estimate of crashes . . . . .	60
9.1	Comparison of grouping algorithms . . . . .	68
9.2	Comparison of prioritization metrics . . . . .	71
9.3	Recovered stack frames from a stack memory dump . . . . .	74
9.4	Continuation of the call graph from Figure 9.1 . . . . .	75
9.5	Variable inspection of an execution trace . . . . .	76
10.1	A selection of the groups created by algorithm 10 . . . . .	88
B.1	!exploitable rules derived from source code . . . . .	113
D.1	Unique crashes in chronological order . . . . .	117
E.1	Call stack grouping of unique crashes . . . . .	119



# Preface

This thesis is written as a part of my degree "Master of Science in Informatics: programming and networks" at the University of Oslo, Faculty of Mathematics and Natural Sciences, Department of Informatics. The thesis is written in collaboration with the Norwegian Defence Research Establishment (FFI) and UNIK University Graduate Center.

## Acknowledgments

I want to thank FFI for letting me write a thesis about a fascinating topic of my personal interest. In particular, I would like to thank my supervisor at FFI, Torgeir Broen, and my supervisor at the Department of Informatics, Audun Jøsang, for excellent guidance and advice.

Others have contributed by draft review and constructive discussions. Trond Arne Sørby, Anders Olaus Granerud and Trond Lønmo have all contributed to the final result.

Last but not least, I would like to thank my family. With their support, the work on this thesis has been an enjoyment.

*Håkon Krohn-Hansen*  
*26th April 2012*

*"Testing shows the presence, not the absence of bugs." [24]*  
*- E. W. Dijkstra (1930-2002)*



**Part I**

**Introduction**





# Chapter 1

## Background

Development of reliable computer software can be a challenging task [74]. History has shown that it is nearly impossible to produce complex software without programming errors. Programming errors can be caused by incorrect design, so-called *design flaws*, but many errors are also caused by incorrect implementation of a correct design. Programs do not always function in the way they were intended. Such programming errors are known as *bugs*. Software bugs can lead to abnormal program termination, a situation commonly known as a *crash*.

### 1.1 Bugs in commercial software

Users and customers have a general expectation that commercial software will function as intended. If this expectation is not met, customers might look for alternative software. Hence, software companies spend considerable resources on testing software in order to find and remove bugs before it is released. Still, some bugs are not discovered until the software has been put into commercial use.

When a weakness in a product is identified, customers expect it to be fixed. A unique thing about software, compared to other products, is that it is possible to distribute fixes automatically to customers at a low cost. A company's ability to improve software by continuously fixing bugs can be crucial to its reputation.

While fixing bugs is important, there is also a public demand for improved functionality in commercial software. Software companies must keep up with technological advances and customer needs. This can imply adding new features that were not invented when the software was first released. Because of practical and economical reasons, developers will try to reuse as much as possible of the original program when implementing new features. A continuous development cycle like this makes it difficult to always base software on a complete and robust design, and there is a possibility that new features will introduce new bugs.

This mechanism can be described as two contradicting forces influencing commercial software. On one hand there is a demand for new software. On the other hand there is a demand for reliable software. The first

adds complexity to software. The second gives stability and robustness.

Considerable effort is put into design and development of operating systems and compilers in order to minimize the risk of bugs. The probability of bugs is reduced by compiler-warnings to programmers about problematic issues. Compilers also add mechanisms to programs to limit the consequence of bugs that are not caught at compile-time. There are even programming languages using so-called *managed code*<sup>1</sup>, hiding low-level features like memory management from programmers. Still there could be programming errors because program code is eventually written by individual developers, and humans make mistakes.

## 1.2 Developing reliable software

Software development goes through many stages, from design and implementation to verification and maintenance. Appropriate actions should be applied at each stage to minimize the amount of bugs. For example, to minimize the occurrence of security related bugs in their software, Microsoft are using a methodology they call *Security Development Lifecycle (SDL)* [65]. SDL is a security assurance process defining best practices in seven phases as shown in Figure 1.1.

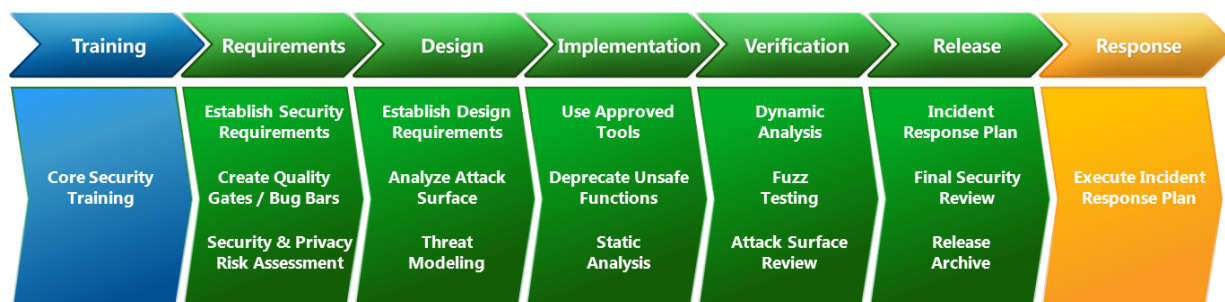


Figure 1.1: Microsoft Security Development Lifecycle (SDL)

SDL focuses on security, but the same mindset applies to assuring stability and robustness. Software customers and consumers expect applications, device drivers and operating systems to be robust and to function without errors and crashes. Incidents of system compromise due to software bugs typically have a negative impact on the customers' trust in the software as well as in the developer of that software. Using a methodology like SDL, software companies try to eliminate bugs as early as possible in the process. During each phase they try to identify and fix potential problems. This includes fixing bugs that are discovered in released software.

### 1.2.1 Software verification using formal methods

*Software verification* [20] is a field of research that aims to verify the correctness of programs. By using *formal methods* [28] it might be possible

<sup>1</sup> "Managed code" is a term used by Microsoft. It is also commonly known as "bytecode".

to generate a complete proof that a given program will always behave correctly. This is especially important for programs dealing with critical tasks such as financial transactions and air traffic control [38]. However, in practice formal methods can only handle software modules of relatively small size.

Recent research has led to the introduction of formal verification in development of commercial software [19]. For example Microsoft provide compile-time static verification tools<sup>2</sup> for driver developers. The tools can detect errors missed by the compiler and by conventional runtime testing. In the context of SDL, compile-time verification is a form of *Static Analysis* while runtime testing is known as *Dynamic Analysis*.

Formal methods have their limitations and cannot be expected to remove all bugs from software. There is for instance a problem with scalability when applying formal methods to complex software systems. This is mainly a problem when trying to prove the correctness of an implementation with respect to its formal specification [51]. A program must work outside a theoretical lab environment.

Formal specification can scale well on its own because formal specification languages like UML<sup>3</sup> [77] offer a high level of abstraction. When formal methods are used to verify an implementation, scalability is limited by the complex characteristics of programming languages and computer architectures. When a program is required to interface with an operating system (OS) and concrete input data it can be difficult to specify the data models and state models necessary to conduct a correct formal analysis. Even if the models could be specified, they would be complex and contain unknown factors.

## 1.2.2 Advances in programming languages

Languages like C and C++ [13] are high-level abstractions compared to machine code, or *native code*, understood by computers. Still they include low-level features that give direct access to the underlying architecture. In particular they leave memory management to the programmer [71]. The programmer is allowed to operate directly on memory addresses through pointers. Pointer arithmetic is the foundation of many classes of bugs [34], such as buffer overflows, null pointer dereferences, double frees and use-after-free bugs. These kinds of low-level bugs are commonly known as *memory corruption bugs*.

Memory corruption bugs are countered by programming languages such as Java [73] and a set of languages used in the .Net Framework [60]. These languages are purely *object-oriented* [72]. The programs are running inside a controlled environment known as the Java Virtual Machine (JVM) and Common Language Runtime (CLR) respectively. This controlled environment handles allocation of memory for objects and performs *garbage collection* of objects that are no longer needed. Implementation can

---

<sup>2</sup>Static Driver Verifier (SDV) [56] and PREfast for Drivers (PFD) [55] are both included in the Windows Driver Kit (WDK) [62].

<sup>3</sup>Object Management Group (OMG) standard: Unified Modeling Language™

be closer to formal specification, which means that formal verification can be applied more easily.

In such programs, memory is managed by the runtime environment and cannot as easily be corrupted. However, these programming languages contain a similar class of low-level bugs which is *insufficient exception handling* [25]. Although the languages do not operate with address pointers, they have object references. If an object reference is not pointing to an object, it has the value null. Dereferencing a null pointer without catching the resulting exception will terminate the program. Incorrect use of an object type can also result in a number of different exceptions which must be handled explicitly. So if programs running native code and managed code are compared from a reliability perspective, they are not different in nature. They both can crash unexpectedly.

Java and .Net provide a safer and more controlled environment for applications. Still most applications on computers use native code because of performance. Tasks that need fast processing of large amounts of data are typically written in C/C++. Another type of software that needs high performance are operating systems. All modern commercial operating systems are written in C and Assembler, which both translate into native code. Device drivers also run as native code, because they operate close to both hardware and the OS.

Another aspect is that the JVM and CLR run as native code. Memory corruption bugs in the managed execution environment could lead to native program crashes triggered from managed code. It is also possible to import native libraries and even launch native programs from the managed environment. Client-side scripting languages like JavaScript [42] also result in the execution of native code by the use of a just-in-time (JIT) compiler.

### 1.2.3 Input validation

Desktop computers are powerful machines performing complicated tasks. Web browsers and document readers are expected to support a number of protocols and data formats. It is infeasible to prove that such complex programs will function as intended given any input. Instead software developers are required to expect the unexpected when developing a program. This applies to both design, implementation and verification.

Expecting the unexpected is particularly important when processing user input. A program should not assume that input data complies to a valid format. This fact has gained importance by the increased use of the Internet, where data often originates from untrusted sources. Missing or incorrect input validation can result in an unwanted situation [34]. The program could stop functioning or do something not intended by the programmer.

Some software bugs can even be exploited by malware<sup>4</sup> to gain complete control of a computer [70].

---

<sup>4</sup>malware = malicious software

### 1.2.4 Software testing

When it is not feasible to prove a program's correctness for all possible input, another alternative is to run the program through a series of tests and observe that it behaves correctly for the test cases [37]. The test cases should include both valid and invalid input. The test runs will prove to some extent that the program will function as intended and possibly reveal bugs. Program testing where a program is automatically given invalid and randomized input is known as *Fuzz Testing* or *Fuzzing* [85].

Fuzz testing has proved to be a cost-effective way to discover bugs before software is released [41]. It can be time-consuming, but since the process can be automated, fuzzing can be performed with minimal supervision, running 24 hours a day. It can continue after release and should be kept running until a new version of the program is ready for testing.

Memory corruption bugs in native code can be identified during fuzzing by detecting program crashes. When a program crash is detected, program memory can be saved in a so-called *crash dump* [88]. A crash dump contains detailed technical information about the state of the program at the moment of crash. A fuzzing run of a million test cases can typically produce thousands of program crashes.

### 1.2.5 Automatic crash reporting

Environmental factors on a customer's computer can produce errors that were not thought of during development or caught during testing. There can for instance be compatibility issues with other software and hardware. To help identifying and fixing program errors occurring in real-life use, software companies can let their programs automatically report back over the Internet when a program crash occurs.

The information reported can include a crash dump together with details about the system configuration and software versions. For commercial software with many users, the amount of crash dumps received can be quite large. For instance Mozilla can receive 2.5 million crash reports from Firefox users in a single day [86]. Microsoft Windows Error Reporting (WER) service is provisioned to receive and process over 100 million error reports per day [39].

### 1.2.6 Program crash analysis

Program crash analysis requires special expertise. The main challenge is to leverage information about a problem at machine level into finding a solution in source code. This is mainly an issue with native code crashes. Managed code is directly linked to source code in a way that greatly simplifies crash analysis.

A crash dump can give information about where in the program code the crash occurred. It can also tell how the program crashed, for example if the program was given an illegal instruction or if an instruction tried to

access an invalid memory address. The *call stack*<sup>5</sup>, which is part of a crash dump, can show the *control flow*<sup>6</sup> leading to the crash. In many cases, the call stack alone can give enough guidance for developers to know where to look for the bug. Most bugs are fixed within the top ten functions of the call stack [78].

However, finding the root cause of a crash can be difficult. It may require deep knowledge and understanding of the program code. Since memory corruption happens on machine level, it may also require understanding of the underlying architecture. This is expertise that developers normally do not have, because they relate to source code.

---

<sup>5</sup> A "call stack" is also commonly known as a "stack trace".

<sup>6</sup> The term "control flow" refers to the order in which different parts of a computer program are executed.

# Chapter 2

## Thesis

### 2.1 Motivation

There are at least two challenging factors concerning program crash analysis. One challenge is to be able to handle the large number of crashes that can be generated from fuzzing and automatic crash reporting systems. This involves identifying similar crashes and prioritizing the crashes according to specific criteria. The other challenge is the complexity of finding a solution that fixes the problem in source code.

#### 2.1.1 Grouping and prioritizing crashes

Millions of crashes cannot be analyzed manually. There is a need for methods that can automatically find relations between crashes. For example several crashes could be caused by the same bug and should be grouped. There is also a possibility that a group of similar crashes can be caused by a number of different bugs.

Another approach could be to prioritize crashes. Common crashes should be fixed before rare crashes. On the other hand a rare crash could be more critical by nature and should be given a higher priority.

#### 2.1.2 Finding the underlying cause of crashes

A crash is only a symptom of a program error. The underlying cause can lie in code located far from the code that generates the crash. A crash dump gives detailed information about a crash, but it does not necessarily tell developers what went wrong and why.

An inherent limitation of a crash dump is that it only shows the state of the program at the moment of crash. The call stack can give important historic information, but it does not show the control flow within functions, nor does it show function calls that have already returned. In a situation of memory corruption there is also a possibility that the call stack is corrupted.

Methods of dynamic program analysis can provide information about execution of the program before the crash. Dynamic analysis can be used as a supplement to crash dump analysis. It could point out where memory corruption takes place. This could take developers one step closer to

finding the cause of the crash. If the corruption can be prevented, the crash will not occur.

## 2.2 Objective

This thesis addresses both identified challenges:

- Strategies for categorization and sorting of crashes are explored.
- Different methods of crash analysis are examined to see if they can help developers find the root cause of crashes and ultimately fix source code.

A set of research questions are used to assess the relevancy of the explored methods. The methods are evaluated by how they contribute to answering the following questions about crashes:

- RQ1: How are crashes related?
- RQ2: How should crashes be prioritized?
- RQ3: How should crashes be fixed?

## 2.3 Thesis layout

The remainder of the thesis is organized as follows:

- Chapter 3:** Explains memory corruption and crash dump analysis
- Chapter 4:** Gives an introduction to fuzzing
- Chapter 5:** Describes methods used for dynamic program analysis
- Chapter 6:** Introduces related research on program crash analysis
- Chapter 7:** Describes the methods used in this thesis
- Chapter 8:** Presents fuzzing results
- Chapter 9:** Presents results from crash analysis
- Chapter 10:** Discusses how crash analysis can help reduce the time needed to locate and fix bugs
- Chapter 11:** Concludes and suggests future work



**Part II**  
**Theory**



## Chapter 3

# Program crash analysis

This chapter outlines what a program crash is and how it can be analyzed. First memory corruption is described in detail. Then crash dump analysis is demonstrated using a simple example program. The example shows benefits and limitations of using a crash dump to investigate a crash.

### 3.1 Memory corruption

Memory corruption in computer software happens if a program operates on data in an incorrect manner. These types of errors can be difficult to investigate. It can lead to subtle and random program behavior. It can exhaust system resources and cause a program to hang or freeze. The best case scenario of memory corruption is actually a crash [44]. A crash is a concrete error that can be investigated. To fully understand memory corruption, it is important to understand how memory is managed on computers.

#### 3.1.1 Protected mode

The most commonly used computer architecture for desktop and laptop computers is the Intel x86 [5]. It is supported e.g. by Microsoft Windows, Linux and Mac OS X. These operating systems are multitasking and rely on *Protected Mode* [81] first available on Intel's 80286 processor in 1982. Protected mode introduced two important features that still are vital to the stability of computer systems. One was hardware support for privilege separation. Another was the isolation of processes by the use of virtual memory.

In protected mode, the CPU<sup>1</sup> can operate in one of 4 privilege levels. The privilege levels are called *rings* and they are named from ring0 to ring3. All modern operating systems use ring0 for system/kernel code and ring3 for user code. Ring0 and ring3 are also referred to as *kernel mode* and *user mode*. In ring3 the instruction set is limited to non-privileged instructions. OS components and drivers can be accessed from ring3 by the use of *system*

---

<sup>1</sup> CPU = Central Processing Unit

*calls*. Transitions between ring0 and ring3 are also frequently performed by the OS because of multitasking.

Multitasking is performed by an OS component known as the *task scheduler*. Multiple tasks can run simultaneously on a single CPU because the task scheduler is constantly switching between the tasks. User processes usually contain one or more running threads, and all threads are scheduled as tasks. At a given moment of time, only one thread is executing code while others are waiting in a scheduling queue. When the scheduler is switching between two tasks, the state of the old task must be saved and the state of the new task must be restored. This is known as *context switching*.

Protected mode was enhanced when the 80386 processor [48] was released in 1985. Some of the enhancements were hardware support for context switching, 32-bit memory addressing and improved memory paging. A 32-bit address bus combined with memory paging allowed operating systems to give 4 GB of *virtual memory* to each process. The memory is called virtual because it does not refer to physical memory addresses. Instead a 32-bit virtual memory address refers to a location in a memory page.

The memory page is located using a two-layered look-up. The 4 GB of virtual memory is divided into 1 M (1024 × 1024) pages of 4 kB each. Figure 3.1 shows a 32-bit linear address as seen by the paging unit. The 10 most significant bits (MSB) is an index into a page directory created by the OS. This gives the address of a page table. The next 10 bits is an index into this page table, which gives the address of the corresponding page. The 12 least significant bits (LSB) contain an offset into the memory page, which gives the physical location of the requested memory.

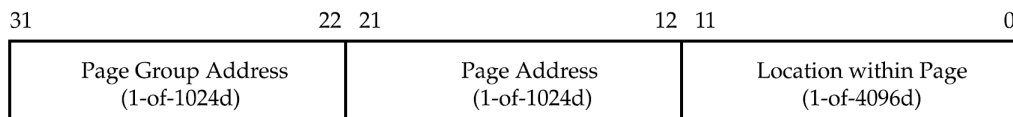


Figure 3.1: Paging Unit's View of 32-bit Linear Address [81]

The requested memory page might be located in physical memory, but it might also be paged out to hard disk in a page file or swap file. This happens to memory pages that have not been recently accessed. If a process wants to access a page that is currently paged out, a *page fault* will be issued. A page fault is handled by the paging unit which loads the page into physical memory so the process can continue. This mechanism is hidden from processes.

Since each process has its own virtual address space, a page directory must be assigned for each process. This is done by using the CPU control register CR3 as a pointer to the page directory of the current process. The page directory is unique for each process, but it is shared between all threads in the same process. Figure 3.2 shows the translation from virtual to physical memory as given in the Intel 80386 Programmer's Reference Manual.

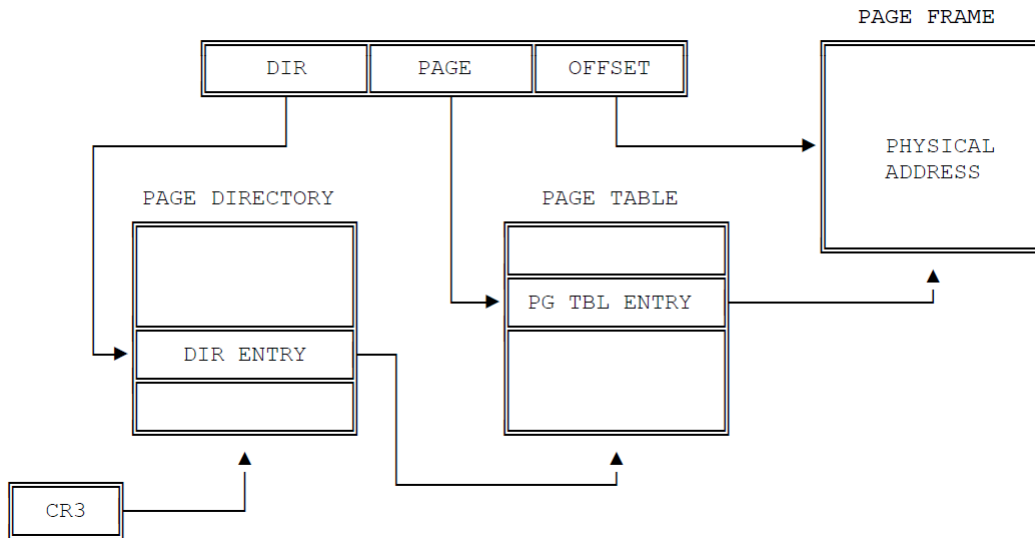


Figure 3.2: Page Translation [48]

Protected mode isolates the memory of each process. One benefit of this isolation is that if a program crashes, it will not crash the whole system. The OS and other programs cannot be directly influenced by memory corruption in one process. Of course a process may cause memory corruption in other processes and the OS by the use of Inter-Process Communication (IPC) and system calls. It can even be triggered from a remote computer over the network. But for that to happen, there must already be a memory corruption bug in the targeted process or OS component. The isolation ensures that a bug in one program can only corrupt the memory of its own process.

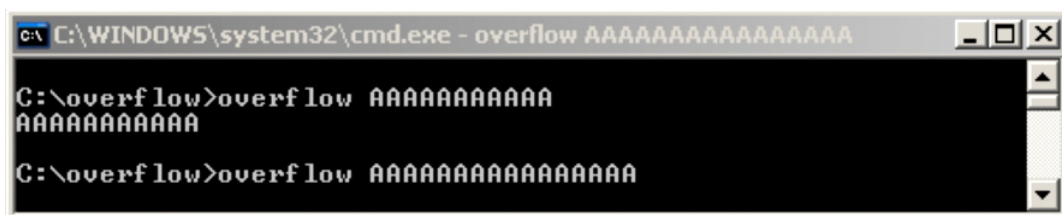
### 3.1.2 Virtual memory

In x86 protected mode each process is given 4 GB of virtual memory. Memory pointers are 32 bits wide and can theoretically address the whole range. However, typically only a small percentage of the 4 GB are valid memory locations. Normally the first 2 GB from `0x00000000` to `0x7fffffff` is user memory and `0x80000000` and above is reserved for the OS.

The user address space of a process is loaded with code and data from the executable file of the program. Any external modules that the program uses must also be loaded into virtual memory. On Windows these are known as dynamically linked libraries (DLLs). All threads are given dedicated memory regions for their respective stacks. Process memory also contains one or more heaps for dynamic memory allocations. In between these memory regions there is unallocated memory. It is inaccessible because it refers to non-existing memory pages.

If a process tries to access unallocated memory, the CPU will normally issue a *general protection fault*. A general protection fault is used if no other exceptions apply. An exception can be handled explicitly by the program

if an *exception handler* is registered. If not, it is handled by the default exception handler which terminates the process. On Windows this kind of exception is known as an *Access Violation* and has the exception code 0xC0000005. Figure 3.3 shows a program crash on Windows XP caused by an access violation.



```
C:\WINDOWS\system32\cmd.exe - overflow AAAAAAAAAAAAAAAAAA
C:\overflow>overflow AAAAAAAAAA
AAAAAAAAAAAA
C:\overflow>overflow AAAAAAAAAAAAAAAAAA
```

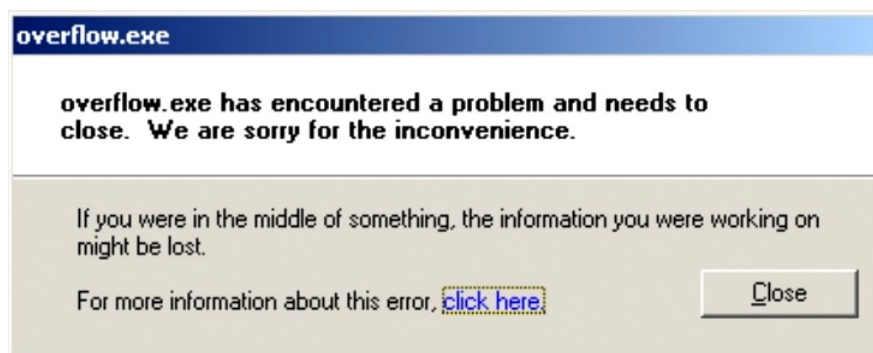


Figure 3.3: Example of a program crash

An access violation can also occur if a process tries to access some allocated memory in an illegal way. Memory protection in modern operating systems is enforced using access rights on pages. Each allocated page is given a combination of the access rights readable (R), writable (W) and executable (E). Originally only R and W were available, but modern CPUs support making pages non-executable (NX)<sup>2</sup>.

### 3.1.3 Corruption of process memory

Physical memory is known as Random Access Memory (RAM). Process memory can be corrupted by hardware errors if bits of RAM are flipped. It can corrupt program data, code or any data in memory. Bit flips on a hard disk may also corrupt process memory. Memory of idle processes are often paged out to disk. If the data is changed on disk and then paged back into RAM, process memory can be corrupted. This is however not the kind of memory corruption that is caused by programming errors. Programming errors cause logical memory corruption.

Logical memory corruption occurs if a program accesses memory in an unintended manner. In the best case scenario, the access is illegal and immediately generates an exception. However, many unintended

<sup>2</sup> Non-executable pages are given different names by different vendors. Microsoft calls it Data Execution Prevention (DEP) [16]

memory operations are legal with respect to page protection. These kinds of corruption may change the state of the program so that it behaves unpredictably and possibly generates an exception at a later point in time.

An example of unintended legal memory access is reading an uninitialized variable. If it is assumed that the variable holds the value zero, it may cause random behavior depending on what the variable is used for. Dereferencing an uninitialized pointer may cause an exception right away. For instance dereferencing a NULL pointer will most likely cause a read access violation. But there is also a chance that the memory location is a valid address. This may result in unpredictable behavior depending on the data at the given address.

Another kind of logical memory corruption happens with incorrect management of dynamically allocated memory. A pointer does not need to be uninitialized to point to invalid data. If a data structure or an object is freed and the pointer is used later, it can point to whatever has been allocated on the same address. This is known as a *use-after-free* bug. If the pointer is freed again, the program will try to free something that should not be freed, known as a *double-free* bug.

Unintentionally overwriting memory is another common problem. This is normally caused by *buffer overflows*. When accessing memory buffers, programmers use pointers and indices. If there is insufficient boundary checking on the pointer values or indices, memory outside the buffer could be accessed. If this memory is read, the situation is similar to reading uninitialized data. The program is given unintended data. If data outside the buffer is written to, it may possibly overwrite other data. Since data buffers normally lie on the stack or the heap, a buffer overflow may overwrite important internal data structures as well as program variables. Again the result can be an exception at some point later when the corrupted data is used.

*Type conversion* can cause memory corruption if they are not accounted for in a program. For example a small negative integer will be interpreted as a huge positive number if it is treated as an unsigned integer. *Integer overflow* can cause the addition of two large numbers to result in a small number. These kinds of programming issues can lead to buffer overflows, for example by the incorrect calculation of a buffer length.

*Concurrency* can also cause memory corruption. If memory is shared between different threads, there can be *race conditions* causing unintended use of memory. This can make one thread corrupt the variables of another thread.

This is not an exhaustive list of possible memory corruption bugs. History has shown that new classes of bugs can be discovered. An example of this is the discovery of *format string bugs*. The bug class was first noted in 1989 during reliability testing of UNIX Utilities [66]. These types of bugs are caused by incorrect use of the C language format string functions like `printf`. The possibility of format string bugs had been present from the development of C, but their implication was not understood until crashes were discovered during testing of real programs.

### 3.1.4 Corruption of kernel memory

Kernel memory is the memory used by the OS. Corruption of kernel memory will in most cases lead to a system crash. On Windows a system crash is known as a *bug check* or a *blue screen* [57]. Memory corruption of kernel memory can be triggered by ordinary users in at least two ways. One is through device drivers and the other is through system calls.

Device drivers running in kernel mode can corrupt kernel memory if they fail to validate input from the device. This can be data from e.g. a USB device or Ethernet data from a network adapter. A memory corruption bug in a device driver can lead to a system crash, which is much more critical than an ordinary program crash. The difference is that a program crash is limited to one process, while a system crash will terminate the whole system and force a reboot. The reboot is necessary because the OS cannot continue running if internal data structures are corrupted.

A system call is a way for user processes to access system services. OS components export a set of functions available for user processes. Similarly device drivers can export functions as a way for processes to communicate with and control a device. When a system call is performed, arguments are provided by the user process responsible for the call. If these arguments are not properly validated, important data structures can be overwritten. This may lead to a system crash. Some of these bugs can also be exploited by malware to obtain system privileges [64].

The next section and subsequent chapters refer to crashes as program crashes only, not system crashes.

## 3.2 Crash dumps

This section shows in detail what information a crash dump can give about a program crash.

Figure 3.4 shows a simple console program written in C. The program takes a text string of input as a command-line argument. The string is copied into a stack buffer (line 21) and the content of this buffer is printed back to console (line 22). The function `stupid_copy` is responsible for the actual copying. The string is first copied into a local stack buffer as an intermediate storage (line 10). Then it is copied from the local buffer to the buffer given by the destination argument (line 11). The executable file `overflow.exe` was compiled with no optimization and without *Buffer Security Check* [23],<sup>3</sup> to make a program as close to source code as possible.

The possibility of memory corruption in the program lies in the two buffers with fixed sizes of 10 and 20 bytes. There are no boundary checks on the length of the input, so the whole string is copied into the buffers. A short input string causes normal behavior of the program, while longer strings will make it crash in at least three different ways. Three scenarios are shown in Section 3.3. Figure 3.3 shows the result of giving the program

---

<sup>3</sup> Visual Studio flags `/Od /GS-`



```
1 // overflow.c
2
3 #include <stdio.h>
4 #include <string.h>
5
6 void stupid_copy(char* destination , char* source)
7 {
8     char local_buffer[20];
9
10    strcpy(local_buffer , source);
11    strcpy(destination , local_buffer);
12 }
13
14 int main(int argc , char* argv[])
15 {
16     char main_buffer[10];
17     char* buffer_pointer = main_buffer;
18
19     if (argc == 2)
20     {
21         stupid_copy(buffer_pointer , argv[1]);
22         printf("%s\n" , buffer_pointer);
23     }
24     return 0;
25 }
```

Figure 3.4: Example C program

an input string of 16 characters on Windows XP. More information about the error is shown in Figure 3.5.

Figure 3.5 shows that the crashing application was `overflow.exe`, and that the crash occurred in the module `msvcr90.dll`. Included in the error signature is the version of the crashing module and application. The module version and offset are important pieces of information. The offset pinpoints the part of the module generating an exception. The exception code is `0xC0000005`, which means an access violation.

The error report contains system information and detailed information from the process memory. The information from memory is what is called a crash dump. It gives information about all the loaded modules, values of CPU registers and raw dump of important memory regions like the stack. If there are multiple threads, the CPU registers and stack memory for all individual threads are dumped.

If the crash dump is sent over the Internet to developers, it can be loaded into a debugger. Then the developers can investigate the crash as if they were using a debugger on the system when the crash occurred. This is known as *post-mortem debugging* [88, 44].

Normally it is impractical to dump the complete process memory. For example Windows can create a so-called *minidump* [44, 39] which contains enough information to investigate most crashes. In case of heap corruption, there is an option of dumping heap memory.

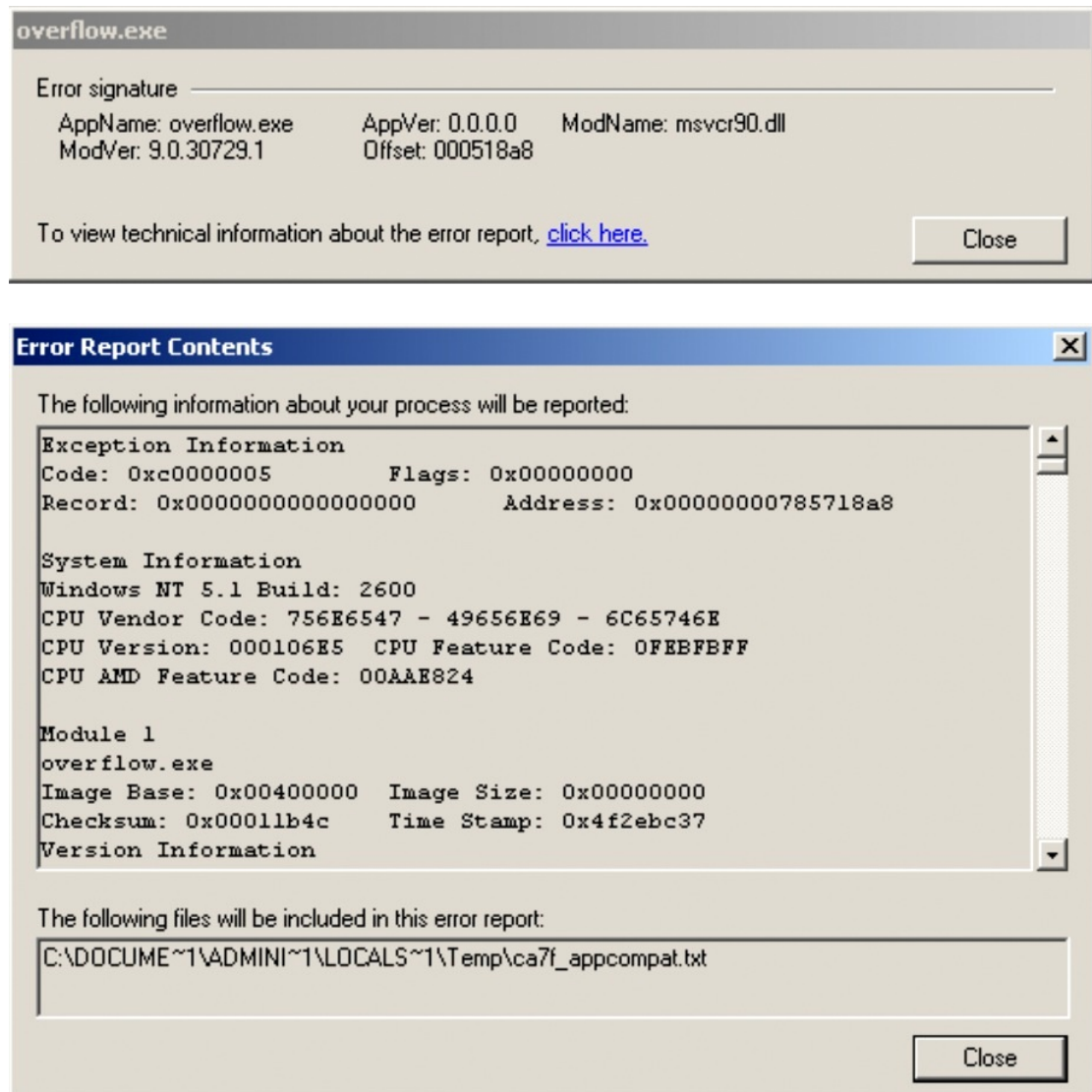


Figure 3.5: Windows Error Report

### 3.3 Crash dump analysis

To understand how memory can be corrupted by a stack buffer overflow, it is necessary to understand the memory layout of the stack. A stack is a last-in-first-out (LIFO) data structure [46] that implements the two operations **push** and **pop**.

In a computer program, each thread is given a memory region to use for its function stack. The stack grows backward into lower addresses. Each function call generates a new *stack frame* which is logically a push operation. When a function returns, its stack frame is logically popped from the function stack. Each stack frame can contain arguments to the function and local function variables. It can contain saved registers so the function does not change these for the calling function. The stack frame also contains the return address so the CPU knows where to resume execution when the function returns. Figure 3.8 shows the stack layout of overflow.exe.

Since the function stack is implemented as a region of virtual memory, it does not only support the push and pop operations. Any memory operation could be performed on the function stack.<sup>4</sup> Programs do this when operating on local variables. Two CPU registers are dedicated as stack pointers. One is ESP<sup>5</sup> which points to the top of the stack. The other is EBP<sup>6</sup> which is the frame pointer. The frame pointer is normally used to reference function arguments and local variables.

```

1 stupid_copy(buffer_pointer , argv[1]);
2
3 00401042  mov     ecx,dword ptr [argv]
4 00401045  mov     edx,dword ptr [ecx+4]
5 00401048  push   edx
6 00401049  mov     eax,dword ptr [buffer_pointer]
7 0040104C  push   eax
8 0040104D  call   stupid_copy (401000h)

```

Figure 3.6: Function call

One can find the memory layout of a specific stack frame by looking at the function call in Assembler code. Figure 3.6 shows the call to `stupid_copy` from `main`. First the arguments to `stupid_copy` are pushed onto the stack in reverse order (lines 5 and 7). When the call instruction is executed (line 8), the *return address* is pushed onto the stack. The return address points to the next instruction after the call. This value will be assigned to the instruction pointer (EIP<sup>7</sup>) when `stupid_copy` returns, so that execution can continue from the correct location in `main`.

The rest of the stack frame is set up in the beginning of the called

<sup>4</sup> NX may prevent execution of stack memory. Still any read or write operation is allowed.

<sup>5</sup> ESP = Extended Stack Pointer

<sup>6</sup> EBP = Extended Base Pointer (The compiler can choose to omit the frame pointer and use EBP as a general purpose register for optimization purposes.)

<sup>7</sup> EIP = Extended Instruction Pointer

function. This code is known as the function *prolog*. Figure 3.7 shows the function prolog of `stupid_copy`. First the prolog stores the previous frame pointer by pushing EBP to the stack (line 3). The frame pointer is then updated to point to the new stack frame (line 4). Line 5 reserves stack space for the local variables ( $14h = 20$ ).

```

1 void stupid_copy(char* destination , char* source) {
2
3 00401000  push      ebp
4 00401001  mov       ebp, esp
5 00401003  sub      esp, 14h

```

Figure 3.7: Function prolog

Figure 3.8 shows the stack frames of `main` and `stupid_copy` after the function prolog of `stupid_copy` has executed. In this figure, the stack grows upward into lower addresses. Hence, ESP points to the lowest address at the top of the stack. When `stupid_copy` was called, a stack frame was created above the stack frame of `main`.

The two stack frames show the space reserved for the two stack buffers. `local_buffer` is given 20 bytes in the frame of `stupid_copy` and `main_buffer` is given 12 bytes in the frame of `main`. Because of the 32-bit architecture, all stack variables are aligned to 4 bytes, giving `main_buffer` two more bytes than specified in source code.

Offset from ESP	Data		Stack Frame
+0x00	local_buffer	Local variables	(lower addresses)
+0x04	...		
+0x08	...		
+0x0C	...		
+0x10	...		
+0x14	stored EBP	previous stack frame	stupid_copy
+0x18	stored EIP	return address	
+0x1C	destination	Arguments	
+0x20	source		
+0x24	main_buffer	Local variables	main
+0x28	...		
+0x2C	...		
+0x30	buffer_pointer		
+0x34	stored EBP	previous stack frame	(higher addresses)
+0x38	stored EIP	return address	
+0x3C	argc	Arguments	
+0x40	argv		

Figure 3.8: Stack layout

The stack layout shows exactly which data will be overwritten if any of the stack buffers are overflowed. When `strcpy` writes to a buffer, it starts with bytes at lower addresses. Hence, a buffer overflow in Figure 3.8

will go downward. If `main.buffer` is overflowed, it will overwrite `buffer_pointer` which is used as an argument to `printf` (line 22). It could further overwrite the stored frame pointer, return value and arguments. A larger overflow would continue to overwrite previous stack frames until the bottom of the stack is reached. An access violation would not be caused directly by the overwrites unless the overflow continued past the bottom of the stack. Then a write access violation would normally occur because this virtual memory address would refer to a non-existing page.

This simple example program can be used to show some of the challenges of crash dump analysis. The following three scenarios demonstrate how crash dumps can be used to investigate program crashes. The call stack is inspected in order to analyze control flow and identify the crash location in source code. Values of local variables and arguments at the moment of crash are inspected with the goal of identifying the cause of the crashes.

The first scenario demonstrates that stack frames of returned function calls are missing from the call stack. The two other scenarios demonstrate how the call stack and crash location can be corrupted.

### 3.3.1 Scenario 1: Distance between corruption and crash

An input argument of 16 characters will overflow `main.buffer` and completely overwrite `buffer_pointer`. This results in an unhandled exception, i.e. a crash, the next time this pointer is used.

Figure 3.9 shows analysis of the resulting crash dump using Microsoft Windows Debugger (WinDbg) [61]. WinDbg shows values of CPU registers and the instruction giving an unhandled exception. In this case the exception is a read access violation. The exception occurred in `msvcr90.dll` as shown in the crash report (Figure 3.5). The failing instruction tests if the EAX register points to a zero byte. The problem is that the value of EAX is not a valid memory address. A corrupted pointer is being dereferenced.

In order to trace back where this corrupted pointer comes from, the call stack can be inspected. The call stack shows all function calls that has lead execution to the crashing location. In this case, the crash was reached from a call to `printf` in the function `main` on line 22 in `overflow.c`. An inspection of the local variables of `main` shows that `buffer_pointer` is indeed overwritten.<sup>8</sup>

To summarize, this crash dump can show where the crash occurred and how the program crashed. It gives a stack trace back to a line in source code of the program. The stack trace can give further information about how the crashing function was reached in a large program. The stack frames can show local variables of all functions that have not yet returned. Knowledge about the program can explain that a pointer variable has been overwritten by a buffer overflow. But the crash dump does not tell where this overflow occurred.

The call to `stupid_copy` has returned and is therefore removed from the

<sup>8</sup> 0x41 is the hexadecimal byte value of a capital A in ASCII encoding.

```
(13c.7c4): Access violation - code c0000005 (first/second chance not available)
eax=41414141 ebx=00403002 ecx=7fffffff edx=785ba173 esi=00000000 edi=41414141
eip=785718a8 esp=0012fc94 ebp=0012ff18 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
msvcr90!_output_l+0x974:
785718a8 803800          cmp     byte ptr [eax],0          ds:0023:41414141=??
0:000> kn
# ChildEBP RetAddr
00 0012ff18 78551e3f msvcr90!_output_l+0x974 [output.c @ 1643]
01 0012ff5c 00401064 msvcr90!printf+0x73 [printf.c @ 63]
02 0012ff7c 004011ce overflow!main+0x34 [overflow.c @ 22]
03 0012ffc0 7c816d4f overflow!__tmainCRTStartup+0x10f [crtexe.c @ 586]
04 0012fff0 00000000 kernel32!BaseProcessStart+0x23
0:000> .frame 2
02 0012ff7c 004011ce overflow!main+0x34 [overflow.c @ 22]
0:000> dv /a
main_buffer = char [10] "AAAAAAAAAAAAAAAA"
buffer_pointer = 0x41414141 "--- memory read error at address 0x41414141 ---"
argc = 2
argv = 0x00332980
```

---

Figure 3.9: Crash scenario 1: 16 characters - Read Access Violation

call stack. In this simple example program, there is only one alternative. The corruption must have happened in the call to `stupid_copy` which also takes `buffer_pointer` as an argument. In a real program the distance between corruption and crash can be much larger. Then it will be nontrivial to find the root cause of the corruption in source code.

In general it is not possible to tell which code has been executed before a crash using only a crash dump. The function stack only shows where functions are called from. The internal control flow within functions is unknown. This control flow may include many function calls which do not show in the call stack. The stack frame of a returned function can be overwritten by any subsequent function call, which means it is no longer in memory.

Even using source code in combination with a crash dump might not show where the memory corruption took place. For example the use of indirect calls of virtual functions can make it impossible to decide from source code which function is actually called at runtime. In the case of a stack buffer overflow, the call stack itself can be corrupted so that no trace could be made back to source code.

### 3.3.2 Scenario 2: Corruption of call stack

An input argument of e.g. 100 characters will overwrite previous stack frames. The crash dump will then show where the crash occurred, but the control flow leading to the crash is missing.

Figure 3.10 shows analysis of the resulting crash dump. This time the crash is a write access violation on the invalid address pointed to by EDI. The crash is traced back to line 11 in the function `stupid_copy`.

```

(7a4.4d4): Access violation - code c0000005 (first/second chance not available)
eax=7efefefe ebx=00000000 ecx=0012ff4c edx=41414141 esi=00000001 edi=41414141
eip=7855b439 esp=0012ff38 ebp=0012ff5c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSVCR90!strcat+0x89:
7855b439 8917          mov     dword ptr [edi],edx  ds:0023:41414141=?????????
0:000> kn
# ChildEBP RetAddr
00 0012ff38 00401023 MSVCR90!strcat+0x89 [strcat.asm @ 178]
01 0012ff5c 41414141 overflow!stupid_copy+0x23 [overflow.c @ 11]
WARNING: Frame IP not in any known module. Following frames may be wrong.
02 0012ffc0 7c816d4f 0x41414141
03 0012fff0 00000000 kernel32!BaseProcessStart+0x23
0:000> .frame 1
01 0012ff5c 41414141 overflow!stupid_copy+0x23 [overflow.c @ 11]
0:000> dv /a
local_buffer = char [20] "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA..."
destination = 0x41414141 "--- memory read error at address 0x41414141 ---"
source = 0x41414141 "--- memory read error at address 0x41414141 ---"

```

Figure 3.10: Crash scenario 2: 100 characters - Write Access Violation

The destination argument is corrupted, and it is natural to suspect that this was caused by line 10. The stack layout in Figure 3.8 shows that an overflow of `local_buffer` can corrupt the input arguments to `stupid_copy`.

Again it is possible to trace the crash back to a line in source code using the crash dump. The cause of the corruption was in the previous line of the same function, but it could have been more difficult to trace. The corruption could have happened in a returned function call, not visible in the call stack. In this scenario the call stack is corrupted so the crash dump does not even show where `stupid_copy` was called from.

### 3.3.3 Scenario 3: Corruption of crash location

In the two previous scenarios, the location of the crash was identified. Figure 3.11 shows a crash dump where both the crash location itself and the call stack is corrupted.

An input string of 27 characters will cause 28 bytes to be copied into `local_buffer` when counting the zero terminator. This will completely overwrite the return address of `stupid_copy` while leaving the function arguments untouched. When the function returns, the instruction pointer (EIP) points to an invalid address giving a read access violation. Since an input of this length will also overflow `main_buffer`, two previous stack frames are corrupted.

Post-mortem debugging of this crash would not yield much results. The memory corruption could have happened anywhere in the program, and the resulting crash could also be anywhere. The program crash would not have to be occurring at the return of a function. A corrupted function pointer could give similar symptoms.

A clue given by the crash dump is that the call stack is corrupted. This

```
(28c.78): Access violation - code c0000005 (first/second chance not available)
eax=0012ff6c ebx=00000000 ecx=0012ff64 edx=00414141 esi=00000001 edi=00403384
eip=00414141 esp=0012ff64 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
00414141 ??                ???
0:000> kn
# ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 0012ff60 0012ff6c 0x414141
01 0012ffc0 7c816d4f 0x12ff6c
02 0012ffff 00000000 kernel32!BaseProcessStart+0x23
```

---

Figure 3.11: Crash scenario 3: 27 characters - Unknown crash location

information could guide developers into looking at stack buffers. Another clue is the pattern of the data on the stack. If the data can be recognized, it might be possible to guess which buffer is overflowed.

### 3.3.4 Limitations of crash dump analysis

The three example scenarios show some possible uses and limitations of crash dumps for investigating program crashes. They illustrate the potential difficulty of finding the link between a crash and its underlying cause.

A crash dump can tell exactly how a program crashed, but it can only give hints about why a particular crash occurred. Useful hints are the crash location and the call stack. They can guide developers to the code that needs review, but a crash dump can never pinpoint the code that caused the crash. The examples also show how a stack buffer overflow might corrupt the call stack and the crash location, removing the most important hints from the crash dump.

If the original input of the program is available, it may be possible to reproduce the crash. Then dynamic analysis of the program can reveal much information that is not present in a crash dump. For example, it can produce a complete *call graph*<sup>9</sup> showing all function calls, including functions that have returned. The call graph can be used to reconstruct a corrupted call stack. If the crash location is undefined, dynamic analysis can also reveal the last known location in the program before the crash.

In a crash reporting system, isolating the original program input can be difficult. Program behavior is influenced by a combination of user input and environmental factors like system configuration, available resources etc. Logging all the input variables that lead to a specific crash can prove to be impossible in practice.

The situation is opposite when crashes are discovered by developers during runtime testing like fuzzing. Then crashes are logged in a controlled environment where all input variables are known.

---

<sup>9</sup> A “call graph” is also commonly known as a “Control Flow Graph” (CFG)



## Chapter 4

# Fuzzing

Fuzzing is a term used for generating random input to a program. The researchers performing the mentioned empirical study of the reliability of UNIX Utilities [66] in 1989 developed a tool they called "fuzz" which generated random characters.

This chapter describes how fuzzing can be an effective way to identify memory corruption bugs. It should be applied in the verification phase of a development lifecycle. Chapter 7 shows in practice how this method can generate crashes in a program for further analysis. Chapter 8 shows the actual results of the fuzzing run performed in this thesis.

### 4.1 Fuzzing strategies

There are several strategies to consider when creating a fuzzer, as described in Table 4.1.

There are two common strategies for fuzzing input to a program. One is called *Generational Fuzzing*. The other is *Mutational Fuzzing*. Generational Fuzzing creates input from a data model such as a network protocol or file format. Mutational Fuzzing takes valid input and performs various mutations on it. A random mutation of an input file is showed in Appendix A. Mutations can also be based on a data model. A detailed data model is what separates a *smart* fuzzer from a *dumb* fuzzer.

An example of an open source fuzzing framework is the Peach Fuzzing Platform [36]. Peach is a smart fuzzer that is capable of performing both generational and mutational fuzzing. Using an XML-file referred to as a *PeachPit* it is possible to define complex data models of the input. The data models affect how the data is being mutated. For example, a number is mutated in a different way than a text string.

#### 4.1.1 Code coverage

An important metric of the efficiency of a given fuzzer is *code coverage*. Ideally a fuzzer should explore as many code paths in a program as possible. Intuitively, if a software bug is to be discovered by fuzzing, the erroneous code must first be executed.

Term	Definition
<b>Dumb fuzzing</b>	Corruption of data randomly without awareness of data structure.
<b>Smart fuzzing</b>	Corruption of data with awareness of the data structure, such as encodings (for example, base-64 encoding) and relations (checksums, bits indicating the presence of some fields, fields indicating offsets or lengths of other fields).
<b>Black-box fuzzing</b>	Sending of malformed data without actual verification of which code paths were hit and which were not.
<b>White-box fuzzing</b>	Sending of malformed data with verification that all target code paths were hit, modifying software configuration and the fuzzed data to traverse all data validations in the tested code.
<b>Generation</b>	Generation of fuzzed data automatically, not based on any previous input.
<b>Mutation</b>	Corruption of valid data according to defect patterns, to produce fuzzed data.
<b>Mutation template</b>	Well-formed buffer that represents an equivalence class of the input. The fuzzer takes the mutation template as an input, producing a fuzzed buffer to be sent to the tested software.
<b>Code coverage</b>	Technology that allows inspection of which code paths were executed during testing. This is useful for verification of test effectiveness and improvement of test coverage.

Table 4.1: Fuzzing taxonomy [71]

Dumb fuzzing by nature gives smaller code coverage than smart fuzzing. The chance of exploring all code paths is larger if the fuzzer is aware of the type of data that is being mutated. To compensate for this, code coverage analysis can be performed when selecting input for a dumb fuzzer. A case study of fuzzing four different commercial applications in 2010 showed that careful selection of input could create a quite effective dumb fuzzer [67].

The input used for mutational fuzzing is known as a *template*. For example when fuzzing a file parser, templates would be valid files of a specific file format. A set of templates can be reduced to a *minimum set* by using code coverage analysis. This is performed by logging which parts of a program that are visited when given the individual template files. A *master template* can be identified by comparing the logs. The master template is the single file that generated the largest code coverage. The log for each input file is then compared to the log of the master template. Only templates that cause exploration of new parts of the program are included in the minimum set.

Peach 2.3.8 includes a tool called `minset.py` [35] which can be used to perform code coverage analysis and create a minimum set of template files. The tool first identifies the basic blocks of the program and then uses Pin [54] to instrument the program and log which basic blocks are visited. *Instrumentation* is covered in detail in Chapter 5.

A basic block is a sequence of code or statements that are not interrupted by branching. A branch can be a jump instruction (`goto`) or conditional jumps created by conditional statements like `if-then-else`. It can be used as an atomic measure for code coverage. If a basic block is reached, then the rest of the block must be executed, unless the code generates an exception. An exception may pass execution to another basic block. This makes logging of basic blocks an accurate measure of code coverage.

Figure 4.1 shows the three basic blocks of the `main` function in `overflow.c` from Section 3.2.<sup>1</sup> There are three basic blocks because of the `if`-statement (line 19). The function `stupid_copy` contains only one basic block, because there are no conditional statements in the function.

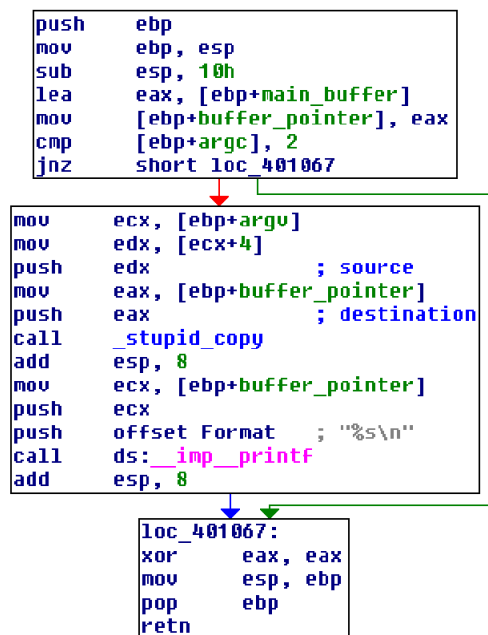


Figure 4.1: Basic blocks of the `main` function in `overflow.c`

The `main` function illustrates how important code coverage is for fuzzing results. If the input string given to `overflow.exe` contains one or more white-space characters, it will never crash the program, regardless of how long the string is. Only a single input argument will be accepted, making `argc == 2`. Any other value of `argc` will make the program jump to the last basic block and return.

<sup>1</sup> `overflow.exe` was disassembled using the Interactive disassembler (IDA) [4]

### 4.1.2 White-box fuzzing

The fuzzing strategies covered so far can all be performed as *Black-box* fuzzing. This means running the program through the tests without actual verification of which code paths were hit and which were not. A more effective approach is known as *White-box* fuzzing [41]. According to Microsoft their white-box fuzzer SAGE<sup>2</sup> found roughly one third of all the bugs discovered by file fuzzing during the development of Windows 7 [40]. SAGE is CPU intensive and is typically run after all other tests. That means that the bugs found by SAGE were missed by all other methods, including static program analysis and black-box fuzzing.

White-box fuzzing involves *symbolic execution* [26] of the program with concrete input. Whenever a branch is reached, an *SMT*<sup>3</sup> *solver* [31, 30] is given the task of manipulating the input so that the other branch is taken. If the solver finds a solution, the program is run again with the new input. This way input is mutated with the concrete goal of exploring new code paths.

## 4.2 Strategies for triaging errors

The previous section showed different strategies for generating invalid input to a program. Common for all the strategies is that they are of no use if program errors are not detected during the test cases. For each iteration of input, the target program could be run through a runtime test specifically made for detecting abnormal behavior. This could for example be useful to validate complex state machines. Memory corruption bugs are normally discovered by detecting program crashes.

During a fuzzing run, a program is likely to crash many times, but it is unlikely that all the crashes are unique. Some of the crashes will occur in the exact same location. Others may be similar although not exactly the same. A fuzzer may use a *bucketing*<sup>4</sup> algorithm [39] to log similar crashes together for analysis. Another aspect is that developers could need help in prioritizing the crashes. Automatic crash dump analysis could classify crashes in categories by severity.

Some of the tools that can be used for automatic bucketing of crashes are !exploitable [59] for Windows, CrashWrangler [89] for Mac OS X and the Valgrind [69] tool Memcheck for Linux and Mac OS X. !exploitable and CrashWrangler are both integrated in Peach.

There is one important benefit from crashes generated by fuzzing. It answers the question "is the problematic code reachable by an attacker?" [29]. Because fuzzing only changes user input, anyone who are allowed to give input to the program, could create the crash. A program crash could also be caused by some internal failure which is not triggered

---

<sup>2</sup> SAGE = Scalable Automated Guided Execution

<sup>3</sup> SMT = Satisfiability Modulo Theories

<sup>4</sup> *bucket* (noun): A collection of error reports likely caused by the same bug.  
*bucket* (verb): To triage error reports into buckets. [39]

by user input. But such a bug would not be as critical to fix as a bug that could allow someone to make the program crash deliberately.

### 4.2.1 !exploitable

!exploitable (pronounced "bang exploitable") is an extension for WinDbg published by Microsoft in 2009. It was created based on the experiences Microsoft had with fuzzing during the development of Windows Vista [29]. It performs both automatic bucketing and prioritization of crashes.

The algorithm that is used by !exploitable to determine the severity of a crash is based on its predicted security implication. All crashes produced by fuzzing can be triggered by an attacker. In that sense there are no false positives, and all the crashes should be fixed [14]. However, some crashes are more likely than others to be exploited by malware to gain control of a computer. The specific kind of exploitation addressed, is the ability to hijack control flow with the goal of executing arbitrary code [70, 15]. Given a vulnerable program, it might be possible to force the CPU to run malicious code injected via user input [17].

Exploitability is determined by a set of rules listed in Appendix B. A crash is placed in a category based on multiple factors. The factors are not shown in the table, but when they have consequence for the classification, they are mentioned in the description in the second column. Important factors are the exception type, register values and code analysis within the basic block in which the crash occurred. The rules are applied in the order of the table. The last column shows if rules are final or not. Rules that are not final may be overridden by another following rule. The last three rows are fallback rules that are used if no other rules apply.

To separate unique crashes from each other, !exploitable creates a hash of the call stack. The hash consists of a major and minor component. The major hash is by default based on the top five stack frames, and the threshold of five is configurable in source code. The offset into each function is not used when calculating the major hash. The minor hash includes the offsets and is also based on all available stack frames.

The minor hash is conservative regarding uniqueness. All stack frames must be exactly equal to produce the same minor hash. The major hash can serve as a bucket for similar crashes. If the function names of the top N stack frames are equal, it will produce the same major hash. This will catch two types of relations between crashes. One type is crashes occurring in different locations of the same function and the function calls leading to the crashing function are the same. The other type is when there are irrelevant differences further down the call stack, but the top N stack frames are equal.

Peach uses the results of !exploitable to group equal crashes. Crashes with equal classifications, descriptions and hashes are treated as equal. In the remainder of the thesis this will be referred to as the unique *name* of a crash.

Although !exploitable was created to triage crashes produced by fuzzing, it can be used to analyze any crash. It relies only on the crash dump, not the input that generated the crash. This would make

it applicable to automatic crash reporting systems where the amount of crashes can be much larger than what can be generated with fuzzing.

### 4.3 Assessing the effectiveness of a fuzzer

A practical question when it comes to fuzzing, is when to stop a fuzzer. When doing white-box fuzzing, it is possible to use code coverage as an indicator. If all program paths have been explored, it is close to giving actual *verification* of the program [40]. In practice it may be infeasible to reach all program paths. Even if all paths are reached, it is not feasible to execute all code with all possible input.

When doing black-box fuzzing, code coverage is not measured. It is only limited by how many iterations of input the fuzzer can create. Depending on the data model, the number of possible inputs could be infinite. The iterations could be created in a deterministic manner. This is the default strategy used by Peach. Then each iteration does only one mutation of one data element and a finite set of iterations is produced. The average time used on each iteration gives an indication of how long it will take to run all iterations. Peach also supports a random strategy which can run forever. The random strategy can also mutate multiple data elements in a single iteration.

#### 4.3.1 Counting unique crashes

An indicator that can be used to assess the progress of a fuzzing run, is the current amount of unique crashes identified. Intuitively, the amount of unique crashes can only increase. Every new unique crash that is identified will increment the counter. For each iteration, the probability that new crashes are unique will decrease. As more and more unique crashes are identified, there is an increased possibility that new crashes will be equal to an already detected crash. Plotting the counter over time should produce a curve that converges toward a horizontal line [67].

A mathematical model of this curve could be constructed by considering balls in a basket. The basket is filled with  $Q$  balls representing all possible inputs to a program that an imaginary fuzzer can produce. Drawing a ball from the basket is the equivalent of a fuzzing iteration. In order to create a simple data model, we assume a constant probability of drawing a given ball. Balls are picked randomly by equal probability and they are put back into the basket after each iteration. This is analog to the random fuzzing strategy of Peach. The deterministic strategy would be the equivalent of picking balls in a given order and never picking the same ball twice.

The balls are of different types marked by integer numbers ranging from 0 to  $k$ . The numbers are not unique, so many balls could have the same number. Picking a ball with the number zero means no crash is detected. There are  $m$  balls in the basket with numbers greater than zero. These correspond to inputs that will generate crashes.

Among the  $m$  balls, there is a distribution of types defined by a set of constants. The amount of balls of a specific type  $j$  is given by the constant

$q_j$  and  $\sum_{j=1}^k q_j = m$ . If one of the  $m$  balls are picked, a copy is made of the ball. The original ball is put back into the basket, and the copy is put in one of  $k$  buckets according to the number on the ball. If all buckets have at least one ball, the fuzzer has produced all crashes that theoretically can be produced under the specific assumptions.

For each new bucket that is filled with its first ball, the amount of unique crashes is incremented. This amount will increase and converge toward a ceiling of  $k$ . Appendix C derives the following expression for the curve of unique crashes:

$$Z(i) = k - \sum_{j=1}^k \left(1 - \frac{q_j}{Q}\right)^i$$

where  $i$  represents iterations. At iteration zero  $Z(0) = 0$ . Appendix C also shows that  $Z(i)$  converges to  $k$  as  $i$  increases toward infinity.

The shape of the curve will depend on the distribution of the constants  $q_j$  and the mutual relations between  $k$ ,  $m$  and  $Q$ . None of these variables can be known for a specific fuzz case, but we can assume  $k \ll m \ll Q$  and an uneven distribution of  $q_j$ . The expression can be used to reason about the shape and characteristics of such a curve. Chapter 8 puts this mathematical model in the context of actual fuzzing results.

Figure 4.2 shows an example curve created by choosing a simple distribution of a small number of unique crashes. The example uses  $Q = 100000$ ,  $m = 2000$  and  $k = 8$ . The distribution of  $q_j$  is shown in Table 4.2.

$j$	1	2	3	4	5	6	7	8
$q_j$	1	2	7	30	60	200	700	1000

Table 4.2: An example distribution of unique crashes

The distribution is chosen so that some crashes are much more common than others. This results in a curve that increases rather quickly in the beginning. After approximately 1500 iterations half of the buckets are filled with balls. Because some of the crashes are rare, the curve does not quickly converge, but rather slowly increases in an almost linear shape toward the ceiling of eight. If there were several rare crashes, this effect would be even more apparent.

If the curve becomes close to horizontal, it is time to stop fuzzing. For example one could consider running a fuzzer for one more week. If at least one new unique crash was detected during the last two weeks, one could argue that the interval between unique crashes is still under 14 days. Then there would be more than 50% chance that a new unique crash would be detected in the next seven days. Bearing in mind that a fuzzer can run without human interaction, it could be cost-effective to keep it running.

When there is a long time interval since the last unique crash, the fuzzer has reached its potential. It has probably found all crashes it is capable of finding. An alternative to ending the fuzzing run could be to change the

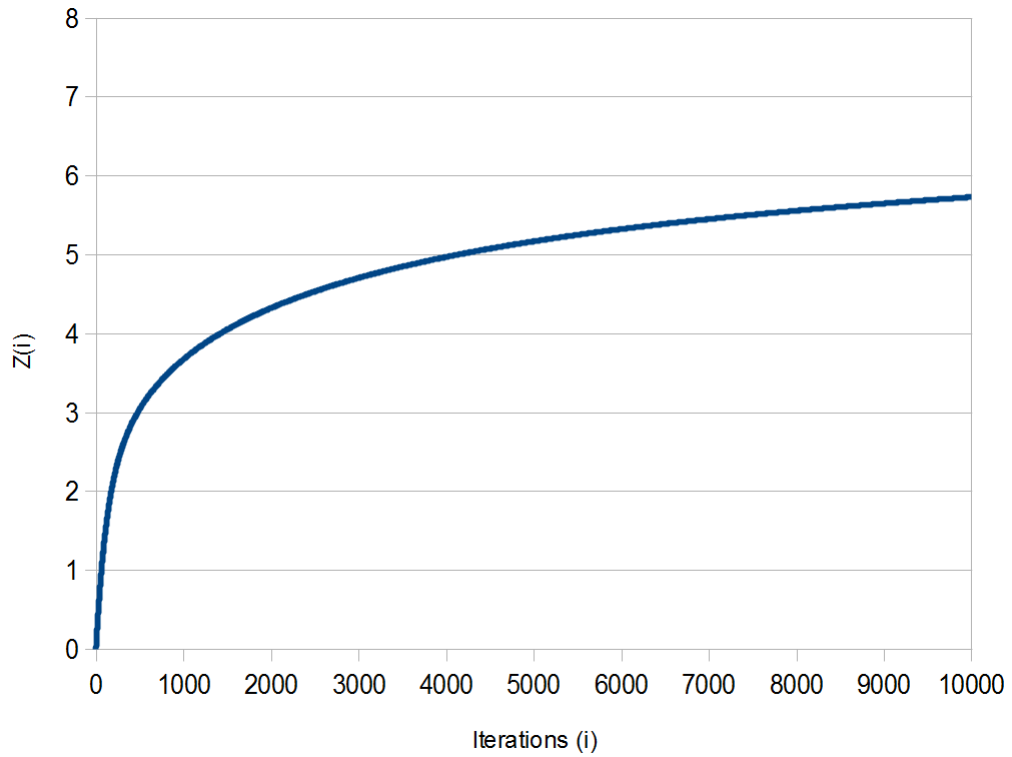


Figure 4.2: Converging curve of unique crashes

fuzzing strategy. This could involve creating a more complex data model or go from black-box to white-box fuzzing. A change of strategy might raise the ceiling, making the curve converge toward a new level.



## Chapter 5

# Dynamic program analysis

A program crash is only a symptom of a program error. Crash dump analysis can be used to investigate program crashes, but there are limitations when it comes to finding the underlying cause of a crash.

As shown in Section 3.3 a crash dump contains the state of the program at the moment of crash. Historic information about execution paths is limited to the call stack. There is also a possibility that the call stack and the crash location is corrupted, so that no direct link can be made to source code.

If the input generating a crash is known or the crash is possible to reproduce, dynamic methods can be applied to further investigate the crash. Dynamic methods can provide information about the *control flow* and *data flow* of a specific execution.

### 5.1 Methods of dynamic analysis

When a program is analyzed dynamically, the program is executed with a specific input. The analysis does not consider all possible execution flows, but rather one specific execution. Dynamic analysis enables inspection of the program state during execution from start to end. It can include pausing and even changing the course of the program. The most common method is to use a debugger.

#### 5.1.1 Debuggers

A debugger can be attached to an existing process, or it can create a new process from an executable file. The debugger will start its own thread within the process. This debug thread is responsible for showing the state of the real threads of the process during execution.

Execution can be paused, e.g. by a user breakpoint or by an access violation. A machine-level debugger can step through single instructions and show the state of the CPU and memory at each step. It is also possible to automatically trace all instructions. This is performed by constant context switching between program threads and the debug thread. The debug thread will then log the instruction pointer at every step.

Example of debuggers are `gdb` [2] for Linux and `WinDbg` [61] for Windows. Both of these are capable of source-level and machine-level debugging.

### 5.1.2 Emulators

Some debuggers are capable of Instruction Set Simulation (ISS). This makes it possible to simulate a different architecture. Another possible use is to simulate execution without actually running the program on the machine. This can be useful for malware analysis.

ISS done by hardware was originally called *emulation*. The term *emulator* is now often used even if it is done by software. Modern emulators like QEMU [6] use *dynamic binary translation* [21] to send chunks of instructions to the CPU. Typically a whole basic block is sent at once. This reduces the need for context switching between the emulated system and the emulator.

### 5.1.3 Dynamic binary instrumentation

When a process is emulated with dynamic binary translation, it is also possible to add some analysis code to the chunk of instructions. This is known as *dynamic binary instrumentation*. After a code chunk is executed, the analysis code is executed independently. This code can be used to inspect some dynamic properties of the emulated process.

Examples of binary instrumentation frameworks are DynamoRIO [1], Valgrind [69] and Pin [54]. Both Valgrind and DynamoRIO are open-source projects. Pin is freely available from Intel.

The power of instrumentation lies in the ability to modify code at runtime. When using instrumentation to create an execution trace, it is not only possible to log the instruction pointer. The value of each instruction operand can be logged so that a trace contains a complete picture of the code executed. Instruction logging gives the control flow, while operand logging gives the data flow of a specific execution.

Since the main purpose of instrumentation is to do runtime program analysis, it is in general more efficient than using a debugger for tracing. While a debugger must be able to pause execution at any time, instrumentation runs continuously. Pin uses a code cache and a JIT compiler which optimizes code and is claimed to give better performance.

The speed depends on the granularity of the instrumentation. For example Pin can instrument per function, basic block or instruction. It also depends on the amount of analysis code which is applied. Instrumentation will typically slow down execution by a factor of 10 to 100. To avoid instrumenting every part of a large program, it is possible to specify where in the program to start and stop instrumenting. When not instrumented, the program will run at normal speed.

### 5.1.4 Whole-system instrumentation

A research project at the University of California, Berkeley has developed a framework for binary analysis called BitBlaze [22]. The dynamic analysis

component of BitBlaze is called TEMU [82]. It is developed on the basis of QEMU, which is a whole-system emulator.

TEMU runs on Linux and is capable of emulating Windows 2000, Windows XP and Linux operating systems. The emulated OS is referred to as the *guest* while the OS that runs TEMU is referred to as the *host*. The purpose of TEMU is to control the guest system by starting processes, giving input and collecting execution traces. This is done by using a command-line interface similar to QEMU.

One challenge of interfacing with processes in a guest OS, is that the emulator sees the physical memory as the guest kernel does. Furthermore physical memory pages can be swapped out to disk. To deal with this, TEMU reads the CR3 register which points to the physical address of the page directory of the current process. For Linux guests, TEMU uses open-source kernel structures to enumerate processes and query process information. For Windows a support driver must be installed on the guest.

For Windows guests TEMU also uses the FS segment register which points to the Thread Environment Block (TEB) of the currently running thread. This pointer is a virtual address as with all process memory. TEMU therefore has to use the page directory to find the correct page table and manually load the physical memory page.

An advantage of emulating the whole OS, is that the instrumentation is really transparent to the processes being instrumented. While ordinary instrumentation modifies code and exists in the same process as the instrumented program, whole-system instrumentation exists outside the guest OS [82]. The only impact on a Windows guest, is that the support driver is loaded and that execution in general is slower than normal. Another advantage of whole-system emulation is that it enables whole-system *taint-tracking*.

### 5.1.5 Dynamic taint analysis

Dynamic taint analysis [79] is a form of data flow analysis. The purpose is to track data as it flows through a program or OS.

In the context of program crash analysis, it can be used to decide if the corrupted memory generating a crash originates from user input or not. Even though a crash can be triggered by user input, it is not necessarily so that the corrupted memory can be controlled by the input data. This is a vital factor to decide the exploitability of a memory corruption bug.

With TEMU, execution traces can be started on any process and input can be given as e.g. keystrokes, network traffic and files. The input can be marked as *tainted*. This makes it possible to see if some data originates from user input or not. Tainted input data is propagated throughout the OS, even though the trace is only made for the target process. The tainted data is for example tracked while being buffered or processed by the OS, but it is only collected as part of the execution trace.

A challenge is how to deal with modifications of data. The data should be tracked even though it is changed by e.g. arithmetic operations. At a given point of execution, a piece of memory may be influenced by several

pieces of input data. For performance reasons TEMU stores only one taint influence per byte of memory.

### 5.1.6 Differential debugging

Traditional dynamic analysis is the analysis of control flow and data flow of one concrete execution of a program. It is also possible to compare different executions. This is known as *differential debugging* [75].

This method can be used to understand how a program behaves in different ways given different inputs. In the case of program crash analysis, differential debugging can be used to compare normal execution with execution that leads to a crash. Specifically, for a crash produced by mutational fuzzing it is possible to compare executions of the program when given the fuzzing template and the mutated input [68].

Two program executions may be compared by call graphs, basic blocks or single instructions. An example of a debugging framework capable of comparing basic blocks of execution is BinNavi [8]. Figure 5.1 shows a comparison of basic blocks reached by two different executions of the same function. Since the release of BinNavi 3.0 in 2010, it is also possible to record data flow of different executions. Instruction operands and the content of involved memory locations can be logged each time an instruction is executed [75].

Similarly, two execution traces produced by TEMU can be compared in order to identify the parts where execution diverges between the traces. A diverging point of execution can be an important factor when trying to understand the underlying cause of a crash created by mutational fuzzing. This is a part of the program that behaves differently when given valid and invalid inputs [68].

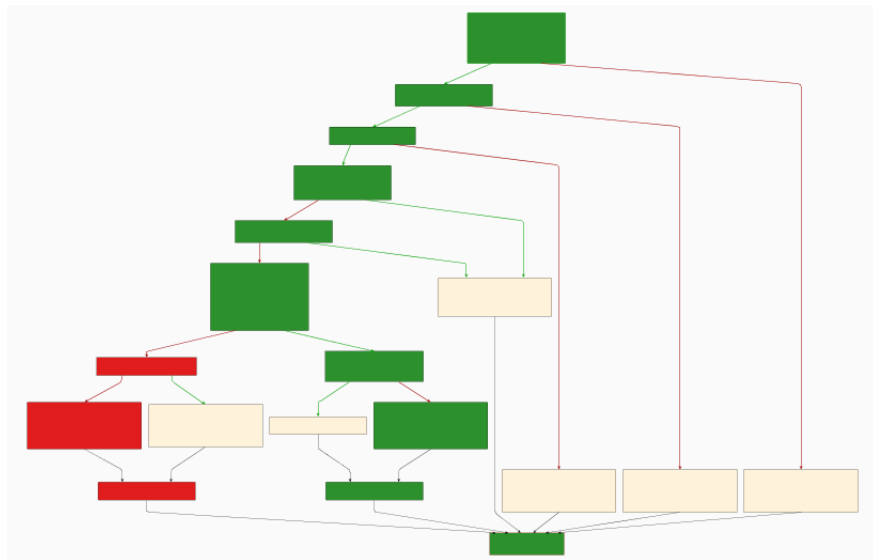


Figure 5.1: Differential debugging with BinNavi

# Chapter 6

## Related research

This chapter gives an introduction to recent research in program crash analysis.

### 6.1 Automatic crash reporting and analysis

Windows Error Reporting (WER) represents by far the largest system for automatic crash reporting [39]. The system is not only available for Microsoft software. Any software, firmware or hardware running together with the Windows platform can take advantage of the service. WER receives crash reports from Windows applications, Windows OS components and third-party device drivers and applications. In addition all JavaScript code running on Windows Live web-sites report crashes to WER regardless of which Internet browser is used.

Third-party desktop application and hardware developers can access the WER service at [sysdev.microsoft.com](http://sysdev.microsoft.com)<sup>1</sup>. The “Windows Dev Center Dashboard” provides vendors with detailed statistics about the stability of their products. Automatic bucketing and analysis of crashes helps developers fix bugs in their software and firmware.

Crash reporting systems can be used not only to catch bugs after release. WER is used by Microsoft during both internal testing and testing through beta-releases. This removes many problematic issues before software is released to the public.

WER contains empirical data about bug fixes and crashes from billions of computers. This data set is unparalleled in both size and completeness. Ten years of experience with WER was summarized in an article in 2009 [39]. The article points out some general aspects of bucketing algorithms.

#### 6.1.1 Bucketing algorithms

An ideal bucketing algorithm manages to assign exactly one bucket for each bug. It also groups all crash reports belonging to a specific bug into

---

<sup>1</sup>The “Windows Dev Center Dashboard” was formerly known as “Windows Quality Online Services” or “Winqual”.

the correct bucket. As stated by Microsoft such an ideal algorithm is not known. Instead a collection of heuristics are applied in order to achieve a form of bucketing that can be useful in practice.

*Labeling* heuristics are performed on clients in order to minimize server load. The labeling consists of identifying meta-information about a crash. This is the type of information shown in Figure 3.5. The label is sent to a WER server which analyzes if additional information is needed. If the label shows a previously known issue, there is no need to request a minidump for analysis. A known solution might be a software upgrade or configuration change. Third-party vendors can even use this service to provide customized troubleshooting messages to customers.

If there is a need for additional information, a minidump is sent together with a detailed technical report containing system information. The minidump is analyzed automatically, using the WinDbg extension `!analyze`. It applies *classifying* heuristics which assigns a bucket to crashes based on how they occurred. A notable result in the article is that the module offset, i.e. the crash location, contributed to the bucketing in almost 68% of the cases.

The article does not mention `!exploitable` as a tool for automatic classification of crashes. A plausible cause for this, is that the article was published in 2009, the same year that `!exploitable` was released to the public. It can provide extended automatic analysis to estimate the severity of a crash and do bucketing based on the call stack. The automatic analysis described serves as an initial analysis for developers. Finding the underlying cause and a fix in source code is a task that requires manual analysis.

There are two complementing types of heuristics influencing the total number of buckets created from a set of incoming crash reports. The first type is *expanding* heuristics. These will increase the amount of buckets with the purpose that no bucket should contain crash reports from more than one bug. The second type is *condensing* heuristics. These will decrease the amount of buckets so that no two buckets contain crash reports from the same bug.

## 6.2 Call stack analysis

The call stack, or stack trace, is an important part of a crash dump. An empirical study conducted by Premraj et al. in 2010 shed some light on the subject. The result of the study was summarized in an article entitled "Do Stack Traces Help Developers Fix Bugs?" [78].

The study analyzed bug reports that contained stack traces. Of the bugs that were fixed, up to 60% of the fixes involved changes to one of the stack frames. Furthermore, it showed that most bugs are fixed in one of the top ten stack frames. It also showed that several stack traces submitted to a single bug report could provide developers with additional information. More stack traces could give multiple perspectives on the same bug.

Stack traces are commonly used to group similar crashes with the goal

of having one group for each bug. Within a group, the different stack traces can be used to understand how the bug is triggered and possibly how it should be fixed.

## 6.2.1 Crash grouping based on call stacks

!exploitable uses a hash algorithm to compare different call stacks. The minor hash can be used to identify strictly unique stack traces. The major hash can be used to group call stacks where the  $N$  top function names are equal. A more detailed comparison of call stacks was suggested by Dhaliwal et al. in 2011 [33].

Their study analyzed crash dumps from users of Mozilla Firefox through the automatic bug reporting system Socorro [86]. A simple, yet effective grouping algorithm was proposed. The algorithm calculates the *Levenshtein distance (LD)* [52] between stack traces. This approach is conceptually different from using application specific heuristics. It is a generic algorithm which can be applied to any set of crash reports containing stack traces.

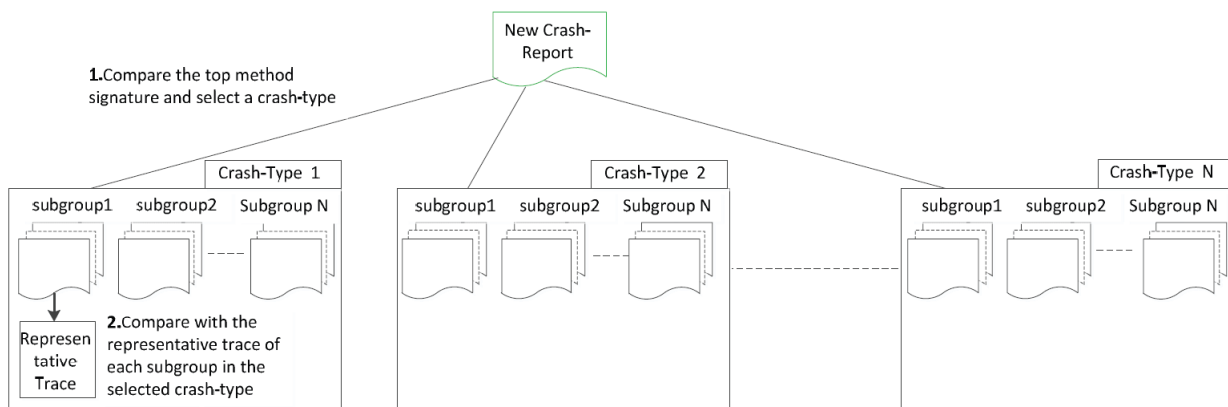


Figure 6.1: A two-level grouping of crashes using representative traces [33]

The Levenshtein distance between two sequences is defined as the number of changes needed to transform one sequence into the other. The legal change operations are *insertion*, *deletion* or *substitution* of a single element. The calculated distance is used to measure the similarity between call stacks.

Within a group of stack traces, it is possible to calculate the average Levenshtein distance between the traces. This is referred to as the *Trace Diversity (TD)* of the group. The Levenshtein distance between different call stacks can be used as a threshold for when a new group should be created. The data set in the study suggested that a threshold of five could ensure that crashes in one group all belonged to a single bug.

The study argues that grouping crashes belonging to different bugs has the worst impact on bug fixing time. Developers are most efficient when given crashes that belong to a single bug. Even if crashes belonging to a

single bug were divided into different groups, the study argues that this is less harmful for bug fixing time than grouping crashes belonging to different bugs.

Two groups represent significantly different stack traces. If two groups actually are caused by the same bug, choosing a stack trace from each group provides two perspectives on the bug, which is shown to reduce bug fixing time. This argument suggests that over-condensing rules in general have more negative impact on effective grouping than over-expanding rules.

The proposed grouping algorithm includes two additional features which are important for the practical implementation in an automatic crash reporting system. One is the idea of *incremental* grouping. The other is that the proposed algorithm uses a *two-level* grouping approach as shown in Figure 6.1. These features are important when considering a large amount of crashes arriving as a continuous stream into the grouping algorithm.

The first level of grouping is based on the top stack frame, which is the crash location. A group of crashes sharing crash locations is referred to as a *crash-type*. The Levenshtein distance is only calculated between crashes belonging to the same crash-type. This considerably reduces the necessary computation. The groups in the second level are referred to as *subgroups*.

In theory, new stack traces could cause reorganization of the subgroups. This is an undesirable effect. Crash grouping should be persistent throughout the bug fixing process. To compensate for this, a *representative trace* is maintained for each subgroup. New crashes in a crash-type are compared only to the representative traces of the subgroups. A crash is put in the subgroup giving the smallest Levenshtein distance. If the distance is larger than the threshold for all representative traces, a new subgroup is created.

## 6.2.2 Call stack graph visualization

In 2011 Kim et al. proposed the use of *Crash Graphs* as a method for improving crash triaging [50].

The proposed graphs give an aggregated view of all stack traces within a group. In such a graph, the nodes represent called functions, i.e. stack frames. The edges of the graph represent the call relations between functions. Both the nodes and edges can be weighted if more than one stack trace contain the same functions or call relations.

There are two proposed applications for such graphs. The first is to identify duplicate grouping, i.e. crashes from two groups belonging to the same bug. The second is to use Crash Graphs to analyze the root cause of crashes. This approach can give developers a more complete view of the context of crashes and possibly show features that a single crash dump cannot reveal.

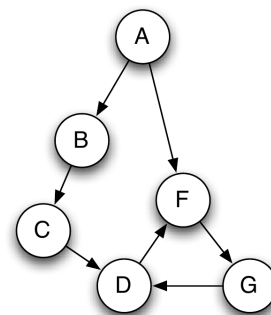


Figure 6.2: Crash Graph [50]



## 6.3 Program crash analysis using execution traces

In 2010 Miller et al. presented their work on program crash analysis using methods and tools from the BitBlaze project [68]. The results were presented through case studies, analyzing execution traces generated by TEMU. The case studies addressed the limitations of crash dump analysis and automatic classification with tools like !exploitable. It was claimed that crash analysis with BitBlaze could help in both assessing the severity of a crash and finding the underlying cause.

### 6.3.1 Crash severity

An important question when assessing the severity of a crash, is if the crash can be controlled by user input.

!exploitable performs taint analysis within the last basic block. The output of this analysis can for example tell if the corrupted data is used as an input argument to a function or if it may be used as the return value of the crashing function. This serves to give a description of individual crashes. The analysis is conservative in the sense that it assumes that all data is tainted.

Since a crash dump only gives the saved state at the moment of crash, !exploitable cannot know if data is influenced by user input or not. For instance if the program crashed during a memory copy operation, the tool assumes that an attacker can control both the source address, the destination address and the length of the copy. This approach may overestimate the possibilities of an attacker.

By using whole-system taint-tracking, the case studies demonstrated how taint information can be included in an execution trace. The important question was if the memory address giving an access violation was tainted. This resulted in reclassification of some crashes.

While this method can give a more accurate classification of crashes, it can be impractical to use as part of a sorting algorithm. The reason is that it typically takes several hours to generate a trace. The proposed working method is to use crash dumps for initial classification and generate execution traces on the most severe crashes.

A limitation of an execution trace is that it does not answer what would happen if there was no unhandled exception. For instance, in the case of a read access violation, the corrupted memory address could have been pointing to an incorrect, yet valid memory region. This could happen by coincidence, or the situation could be produced by a technique known as *heap spraying* [83]. The technique works by allocating several large chunks of memory. This decreases the probability that a given random address is invalid.

If execution were allowed to continue, the memory corruption would most likely result in a later crash. Such a situation could effectively turn a read access violation into a write access violation, which is generally classified as more severe.

A factor related to severity, is the *reliability* of a crash. A crash that

occurs every time given the same input, is more critical to fix than a crash that is more random [32]. Random crashes are not guaranteed to occur during dynamic analysis, which makes them more difficult to analyze dynamically. An execution trace can therefore be even more important in these cases. If an execution trace of a rare crash is successfully produced, the bug could be analyzed and fixed without the need to trigger the crash again.

### 6.3.2 The underlying cause of crashes

An execution trace generated with TEMU contains operand values so that both control flow and data flow can be analyzed. With this information, an analyst can step forward and backward in the code and always know why the program behaved the way it did, at least on the machine level. This reduces the need for additional dynamic analysis.

An alternative is to run the application through a debugger, set breakpoints and step through code manually to inspect the program state. A problem is that it is not always obvious to know where to set breakpoints. Also the problematic code can be run several times before leading to a crash. Manual dynamic analysis can therefore be a long process of trial and error.

An advantage with BitBlaze that is pointed out, is the integration of dynamic and static analysis methods. This is illustrated by the use of static analysis on execution traces. A method called *slicing* can be performed on an operand to filter out the instructions earlier in the trace that may influence the value of that operand. This could help an analyst to focus on the code that was important for the crash in a large program.

Differential debugging can be performed by comparing traces. The case studies demonstrated *aligning* of a good trace produced by using a fuzzing template and a bad trace produced by using the mutated input. A trace alignment is a matching between instructions in two traces, while preserving the ordering of the instructions. The result of an alignment shows the parts of the traces that are equal, and where there are differences. This could pinpoint where control flow diverges. At the diverging points, data flow analysis may answer why the bad trace takes a different path than the good trace.

In 2011 a method called *differential slicing* [49] was proposed to further isolate the differences between two executions. This method outputs a *causal difference graph* showing relations between differences in input (cause) and differences in execution (effect). Another application for differential slicing is analysis of malware with environmental-dependent behaviors.

TEMU is also capable of recording information about dynamic memory allocations and deallocations. Memory addresses in a trace can then be queried for information about where it was allocated and the size of the allocation. If an allocation is not found, the allocation closest in memory is shown. This could be useful to understand heap buffer overflows and other bug classes involving dynamically allocated memory.

### 6.3.3 Time and cost issues

One reason why fuzzing has proven to be effective, is that it can run without human interaction. If a task can be automated and distributed over a cluster of computers 24 hours a day, it can be less expensive than the human resources needed to perform the task manually. Even if a human expert could perform the task quicker than computers, human expertise is in general more expensive per hour, and the amount of work hours of humans per day is limited.

Miller et al. [68] argues that the same logic could apply to crash analysis using execution traces. In the case studies, generating tracing could take hours. Slicing could also take a couple of hours to complete, depending on how far back it was possible to slice. While these automated tasks take time, they can be useful when analyzing many different crashes. Tracing and automatic trace analysis can run in the background while an analyst is performing manual analysis of already produced traces, slices and alignments.

Others have also suggested that semi-automization is more productive than trying to automate a whole process involving program analysis. Routine aspects of the process should be automated giving an analyst the freedom to focus on certain key portions. This has been stated in the context of both automatic vulnerability detection [43] and automatic exploitation [32]. These areas are related to program crash analysis and bug fixing, because they may require understanding of programs at machine level. The methods must also be able to scale to real-world programs.



**Part III**  
**Methods**



## Chapter 7

# Planning the thesis

The goal of this thesis is to explore how software developers can benefit more from program crash analysis. Specifically, it addresses the challenges developers face when presented with thousands and millions of crash dumps.

### 7.1 Methods overview

Existing research on automatic crash reporting systems suggests how crashes can be grouped in order to reduce bug fixing time [33]. Strategies for prioritizing crashes has been proposed in the context of fuzzing results [59, 68]. Dynamic binary instrumentation can give additional information about a crash. This information is claimed to help in both prioritizing and finding the root cause of crashes [68].

#### 7.1.1 Data collection methods

In order to evaluate the different claims, this thesis analyzes a data set consisting of real crashes from a real program. Before the analysis could be performed, a set of crashes was generated. The crashes were produced by using black-box, dumb, mutational fuzzing [36].

There are some advantages of analyzing fuzzing results compared to field crash reports:

- The program version and system configuration can be controlled, so that this is the same for all crashes.
- The data set includes the program input producing each crash.

An automatic crash reporting system may contain crash dumps from several major and minor releases of a program. This generates the need for signatures of stack frames which are invariant between program versions [33]. In fuzzing results of one program version, a stack frame consists of a module, function and offset which makes it trivial to identify equal and different stack frames.

When the program input of each individual crash is known, it is possible to reproduce crashes. This makes it possible to measure the

stability, or reliability, of crashes. Such an analysis can answer if a crash occurs every time, or if it was a random and rare event. Some invalid input might even produce different crashes on each program run. This can only be discovered if the original input is available. Another advantage of having the original input, is that it can be used to perform dynamic analysis of the crash.

There can also be disadvantages of analyzing fuzzing results compared to field crash reports:

- The data set might be smaller than a set of field crash reports.
- Information about bug fixing time is not included.
- The actual links between crashes and bugs are missing.

The amount of crashes that can be generated by a simple fuzzing run can never be as large as the crash databases of commercial crash reporting systems like WER [39] and Socorro [86]. To compensate for this to some extent, the target program and fuzzing strategy is chosen with the goal of maximizing the amount of different crashes.

It is outside the scope of this thesis to actually find a source code fix for all crashes produced. Consequently, only assumptions can be made about bug fixing time and the links between bugs and crashes. The analysis will rely on the results of existing research when it comes to criteria for effective grouping and prioritization.

### 7.1.2 Data analysis methods

The crash dumps created by fuzzing are analyzed with the objective of grouping and prioritizing the crashes:

- Different grouping algorithms based on stack traces are explored. Levenshtein distance is used to measure the effect of different grouping algorithms.
- Automatic classification, frequency and reliability are criteria used for prioritizing crashes.

Crash dumps can be used to uniquely identify, group and prioritize crashes, but they do not contain information about the program states prior to the crash. This thesis uses dynamic binary instrumentation in three different ways:

- Corrupted call stacks can be reconstructed, providing more correct information to the grouping algorithms.
- Taint-analysis can provide additional information for classification and prioritization of crashes.
- An execution trace with operand values can help identifying the root cause of a crash.



## 7.2 Choosing a target program

The target program was chosen based on several criteria. It should be a program simple enough to be suitable for fuzzing and complex enough to produce a large set of different crashes within the time frame of this thesis.

### 7.2.1 A program suitable for quick fuzzing

A command-line file processor is a suitable target for fuzzing. It takes a file as input, does something based on the content of the file and then exits. If the program exits without crashing, there are two alternatives. Either the file contained valid data, or the program dealt with any invalid input in a proper way.

The goal of a fuzzing run is to produce corrupt input which will make the target program crash. To produce many different crashes, the program must be run several times with different input files. If a crash occurs, information about the crash is collected as well as the corrupted input file. This process will in general go faster with a console program than a program with a graphical user interface.

### 7.2.2 A program liable to contain memory corruption bugs

A suitable file processor would be one that parses a complex file format. A binary file format would be preferred, because it in general gives a more compact data representation. This makes dumb fuzzing mutations like bit-flipping more probable of exploring new code paths.

An old program is preferred over a newer program. Advances in compilers may have introduced mechanisms that will make memory corruption bugs more rare. Early versions of a program are also in general more immature, while newer versions have been made more robust by applying multiple bug fixes.

### 7.2.3 Criteria

In addition to the described criteria, it is natural to choose an open source project, so the analysis can be performed from the perspective of developers. The criteria for the target program can be summarized as follows:

- it is a console application (command-line utility)
- it is small
- it is an early version of a program
- it has some specific, complex functionality
- it reads a file as input and processes the file content
- it is open source

## 7.2.4 The target - GNU Ghostscript 6.51

The choice for a target program is GNU<sup>1</sup> Ghostscript [12], an open source program that can be used to interpret PostScript<sup>TM</sup> [47] files (.PS). Ghostscript includes a command-line utility which can be used to convert a PostScript file into the Portable Document Format (PDF) [11]. This is a task that requires complex parsing of the input file and the generation of many complex data structures for the output file. The conversion is executed by the command: `gswin32c.exe -dNOPAUSE -dBATCHE -sDEVICE=pdfwrite -sOutputFile=outputfile.pdf inputfile.ps`

Ghostscript is written entirely in C/C++, and supports many platforms, such as Microsoft Windows, Apple Mac OS, Linux and Unix systems. It is copyrighted by Artifex Software, Inc. Before 2004 Ghostscript was licensed under Aladdin Free Public License (AFPL). The current version is licensed under GNU General Public License (GPL). Version 9.04 for 32-bit Windows was compiled 2011-08-05. Older binary releases are available for download.

The oldest binary release available is GNU Ghostscript 6.51, compiled 2001-07-31 [10] for Windows. Using the original binary release for fuzzing increases the probability of producing many different crashes. This version of the program is not only created from old source code. It was also created by an old compiler. Even if fuzz testing was known at the time, it is unlikely that the program from 2001 was sufficiently fuzz tested. Advances in computation speed over the last decade also increases the chance of producing a suitable data set for the analysis.

The program consists of the executable file `gswin32c.exe` and the library `gsdll32.dll`, which are both found in the directory `gs\gs6.51\bin`. The conversion also depends on some files in the directories `gs\gs6.51\lib` and `gs\fonts`. These directories are kept unchanged as created by the default installation. The code that does the actual parsing and conversion is in `gsdll32.dll`. The program uses three third-party libraries which are statically linked. These are:

- jpeg 6b (1998-03-28)
- libpng 1.0.8 (2000-07-24)
- zlib 1.1.3 (1998-07-10)

## 7.2.5 Symbols

One disadvantage of using the original binary release, is that the original debug symbols are not available. When compiling a program for Windows, a Program Database (PDB) [58] of each executable file can be created. This file includes names of functions and variables. In particular, it can be used to link binary program locations to lines in source code.

To compensate for this, the source files were compiled into new executable files using the correct version of all third-party libraries. Then

---

<sup>1</sup> The name "GNU" is a recursive acronym for "GNU's Not Unix!"

the Interactive Disassembler (IDA) [4] was used to create a program database (IDB) for the original version of `gsd1132.dll` and the new version which included a PDB.

An IDA plugin called BinDiff [7] can be used to port symbols from one program database to another. With this tool, symbols were ported from the IDB with symbols to the IDB file of the original library. This provided function names and variable names for static analysis of crashes at machine level. The symbols were used to manually find links between binary program locations and source code. However, dynamic analysis with WinDbg did not benefit from these symbols.

When debugging DLL-files without a PDB file, WinDbg uses the export symbols defined in the DLL. These contain the names of all functions exported by the DLL. Since !exploitable is a WinDbg extension it also requires a PDB file to correctly display stack frames. To compensate for this, the stack traces were post-processed using a function mapping exported from the IDB. The major and minor hashes were also recalculated based on the new stack frames.

### 7.3 Fuzzing strategy

A discussion of the choice of fuzzing strategy does not directly answer any of the research questions asked in Chapter 2. However, it documents and explains how raw material was produced for further analysis. As shown in Chapter 8, the fuzzing run gave a large set of different crashes, which made the analysis interesting.

The choice of fuzzing strategy depends on both the target program, the input format and available resources. In the case of the PostScript-format, the data model is quite complex. This makes a dumb fuzzing strategy much simpler to implement. White-box fuzzing would probably produce more crashes than black-box fuzzing [40]. However, previous research has shown that random black-box testing can be effective against real programs [37, 67]. Black-box fuzzing is simpler to implement, and there are several available fuzzing frameworks that are capable of doing black-box fuzzing.

Peach Fuzzing Platform [36] was chosen for this thesis. It is capable of performing black-box mutational fuzzing using predefined mutators on a specified data model. It also supports using !exploitable for automatic classification and bucketing of crashes. When using Peach to fuzz PS-files, it is possible to define two different simple data models for the files.

The simplest data model defines the whole file as one piece of binary data called *Blob*<sup>2</sup>. Since PS-files contain mostly text, it would also be natural to use a data model which defines the whole file as one *String*. While these data models are simple, Peach makes use of several different mutators to generate changes to the data. When using such simple data models, code coverage will strongly depend on the set of files used as fuzzing templates.

---

<sup>2</sup> Blob = binary large object

### 7.3.1 Creating a set of fuzzing templates

The fuzzing templates should contain different kinds of features, so that different parts of the target program would be explored. Instead of generating a large set of files based on a data model, it would be more efficient to download a large set of real-world PS-files from the Internet. The set should be a random and representative selection of files.

The website [www.FindFiles.net](http://www.FindFiles.net) has indexed over 700 million files of different file formats, including PS. By searching for a keyword, up to 300 download links are listed. It was possible to download 10.000 random files in a work day by selecting random keywords from a word list and automatically download all files listed. A file size limit of 2 MB was set in order to restrict the computational complexity of both code coverage analysis and fuzzing.

The files were reduced to a minimum set based on code coverage by using the tool `minset.py` [35] released with Peach. The code coverage analysis making coverage traces for all input files can be run in parallel. This analysis was run on eight CPUs over a period of one week producing coverage traces for over 6000 files. The analysis also provided a master template, which is the file that produced the largest coverage of unique basic blocks. Comparing all traces resulted in a minimum set of 369 PS-files.

### 7.3.2 Fuzzing procedure

Peach was run in parallel on eight virtual machines (VMs) [87]. First all files from the minimum set were fuzzed sequentially using the Blob data model. While the smallest files could be run with all iterations, a limit of 20.000 iterations was set for the larger files. This limit was based on a calculation of average iterations per day, and that all files should be fuzzed within a reasonable time frame.

After the sequential run was finished, the random fuzzing strategy was applied with the String data model. Then Peach would use both the Blob-mutators and the String-mutators. The random strategy was configured to switch input files randomly after 15.000 iterations. It could also do a maximum of seven mutations to each file at once.

Peach automatically gathers information about crashes and stores it in a log directory. A new directory is created for each instance of Peach. When fuzzing files sequentially, each new fuzz template creates a new instance. Crashes that are considered equal, are automatically grouped, but only if they were detected during the same instance. A shared root directory was used as log directory for all the VMs. This directory tree became the raw material for the further analysis of crashes.

## 7.4 Crash reliability analysis

A crash found by fuzzing should be reproduced by running the test case again [71, 32]. Each new unique crash should be verified. This can

determine the reliability of the crashes.

A single corrupt input could potentially produce a number of different crashes depending on how deterministic the program is. Even if some crashes are not verified, they should be saved for analysis. They could be related to other crashes or give clues on a bug that was not easily triggered by fuzzing.

In general, a crash that occurs every time by a given input, is more critical than a crash that occurs less often. On the other hand an unreliable crash can be the symptom of a subtle program bug. Investigating such program crashes may be important to enhance the stability and robustness of programs.

The questions asked for each crash in the verification process are:

- Is the crash verified/reproducible?
- How often is the crash reproduced?
- Does the same input produce different crashes?
- Are there produced any new unique crashes?
- Are there produced any crashes already registered?

The verification process was automated in such a way that the input of all unique crashes could be given  $N$  times to the target program. The results were automatically compared with the original crashes. If the verification produced different crashes than the original, the new crashes were automatically compared to other unique crashes. This could identify previously unknown relations between different crashes.

## 7.5 Comparison of grouping algorithms

Grouping algorithms are compared using the methods of Dhaliwal et al. [33] To be able to compare results, Levenshtein distance (LD) and trace diversity (TD) is calculated as described in their article. LD is based on the top ten stack frames, using only the function name. TD of a group is the average LD of call stacks in a group. A low TD means a high similarity within the group.

Silhouette validation [76] is also used to evaluate the grouping generated by the different algorithms. A Silhouette value ( $S$ ) is a value between  $-1$  and  $1$  which measures how well a data set is grouped. For a given crash, a silhouette value close to  $-1$  implies that the crash should belong to another group. A value near zero indicates that a new group should be created for the crash. A value close to  $1$  means that the crash belongs to the correct group.

For a crash  $i$  the silhouette value  $S(i)$  is defined as:

$$S(i) = \frac{b(i) - a(i)}{\max(b(i), a(i))}$$

where  $a(i)$  is the average dissimilarity between the crash  $i$  and all other crashes in the same group, and  $b(i)$  is the smallest average dissimilarity between  $i$  and all crashes in other groups. To gain a positive  $S(i)$ , a crash should be more similar to the crashes within its own group than to crashes in other groups. Similarity is measured by the LD between call stacks. A higher LD means a larger dissimilarity.

Silhouette values were used by Dhaliwal et al. to validate that their two-level grouping approach produced a well-clustered set of groups. However, in the calculation of  $b(i)$  they compared crashes only to other subgroups of the same crash-type. This might overestimate how good the grouping is, because it will not account for similarities between groups of different crash-types. In this thesis, silhouette validation is therefore based on comparison with all other groups.

Silhouette values are only calculated for groups with more than one unique crash. Otherwise,  $a(i)$  would be zero and  $S(i)$  would be 1 regardless of how similar the crash is to crashes in other groups.  $S(i)$  can also not be calculated for a single group, because other groups are needed to decide the value of  $b(i)$ .

## 7.6 Dynamic analysis

BitBlaze [82] was chosen as a framework for performing dynamic analysis of crashes. Some equivalent analysis could possibly have been done by using a debugger like WinDbg or another instrumentation framework like Pin. However, BitBlaze provides ready functionality that may serve the research questions of this thesis.

The TEMU plugin `tracecap` can produce execution traces that log all executed instructions of a process, including the values of instruction operands. Furthermore, the traces show taint information propagated from tainted input and into the process memory. All this can be performed without any modification or addition to the freely available code [22].

The traces are saved in a binary file format. These files can be read by the program `trace_reader` to show the information in text format. The same program can also be used to generate a complete call graph from an execution trace.

The methods utilizing BitBlaze in this thesis are based on the paper “Crash analysis with BitBlaze” [68] by Miller et al.

**Part IV**

**Empirical results**





## Chapter 8

# Fuzzing results

This chapter analyzes the results of the fuzzing run described in Section 7.3. After fuzzing, the reliability of crashes was measured by the crash verification process described in Section 7.4. First the concrete output from fuzzing is described. Then the results of the reliability analysis is presented.

### 8.1 Crash statistics

The fuzz testing of Ghostscript 6.51 ran over a period of seven weeks. In this period, over 5000 crashes were collected in almost 700 log directories. Crashes were automatically grouped and classified by !exploitable. Unique crashes were sorted chronologically and plotted in a graph in order to assess the progress of the fuzz testing. In addition, the instruction pointer (EIP) of all unique crashes were compared, giving a set of unique crash locations.

#### 8.1.1 Automatic grouping and classification

Automatic grouping and classification by !exploitable gave 62 unique crash names listed chronologically in Appendix D. The unique names given by Peach consist of elements from the analysis done by !exploitable, i.e. classification, description and call stack hashes. In addition, each unique crash was assigned a three-digit number for reference.

In the set of 62 unique names, there were 61 unique call stacks. The reason is that two unique crashes had equal call stacks:

```
005_PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x13554811_0x4f4a1368
021_UNKNOWN_ReadAV_0x13554811_0x4f4a1368
```

Their hashes are equal, but they were given different names because of register values. 005 is described as ReadAVNearNull while 021 is given the description ReadAV.<sup>1</sup>

The stack frames in the crash dumps are given as offsets into functions of an executable module, on the form module!function+offset, or

---

<sup>1</sup> AV = Access Violation

module+offset if symbols are not available for the module. The base address of a module can change between program instances. This is especially the case if ASLR<sup>2</sup> is used [53]. That makes an offset a more reliable reference than a virtual address.

The top stack frame gives the location in a module where an unhandled exception occurred. From the data set produced, it was found that all crashes occurred in `gsd1132.dll`. The version of the module and the base address were the same for all crashes. This gives a one-to-one relationship between the top stack frame and the virtual address of the failing instruction (EIP). The terms *crash location* and *EIP* are therefore used interchangeably in the remainder of the thesis.

Since the top stack frame is the same as the crash location, the number of unique EIPs can only be less than or equal to the number of unique call stacks. The minor hash calculated by !exploitable is depending on the complete call stack, including the crash location. Following from the hash algorithm, a new EIP will generate a new unique call stack hash.

The number of unique crash locations was 29. In the two-level grouping approach described in Section 6.2, the EIP groups would correspond to crash-types. Alternatively, if only function names were used, it would give 21 function groups. There were large variations between the EIP groups. About two thirds of the groups had exactly one call stack. The remaining third had more than one call stack. The largest group had 19 different call stacks.

Table 8.1 gives a summary of the classifications given by !exploitable. About 71% of the 62 groups of unique crashes were classified as Unknown. That is 87% of the total crashes shown in the third column. Only six unique crashes with five different EIPs were classified as Exploitable or Probably Exploitable. That is under 10% of the unique crashes.

Classification	Unique	Total
Exploitable	3	3
Probably Exploitable	3	84
Unknown	44	4758
Probably Not Exploitable	12	617
<b>Sum</b>	62	5462

Table 8.1: Severity estimate of crashes

### 8.1.2 Analysis of the fuzzing progress

In the log directory tree, each instance of Peach performed individual bucketing of the crashes detected during that instance. To keep track of the fuzzing progress, all crashes were collected from the log tree and sorted chronologically.

The chronological set was used to identify when each unique crash name was first detected. At this point, the counter of unique crashes

---

<sup>2</sup> ASLR = Address Space Layout Randomization

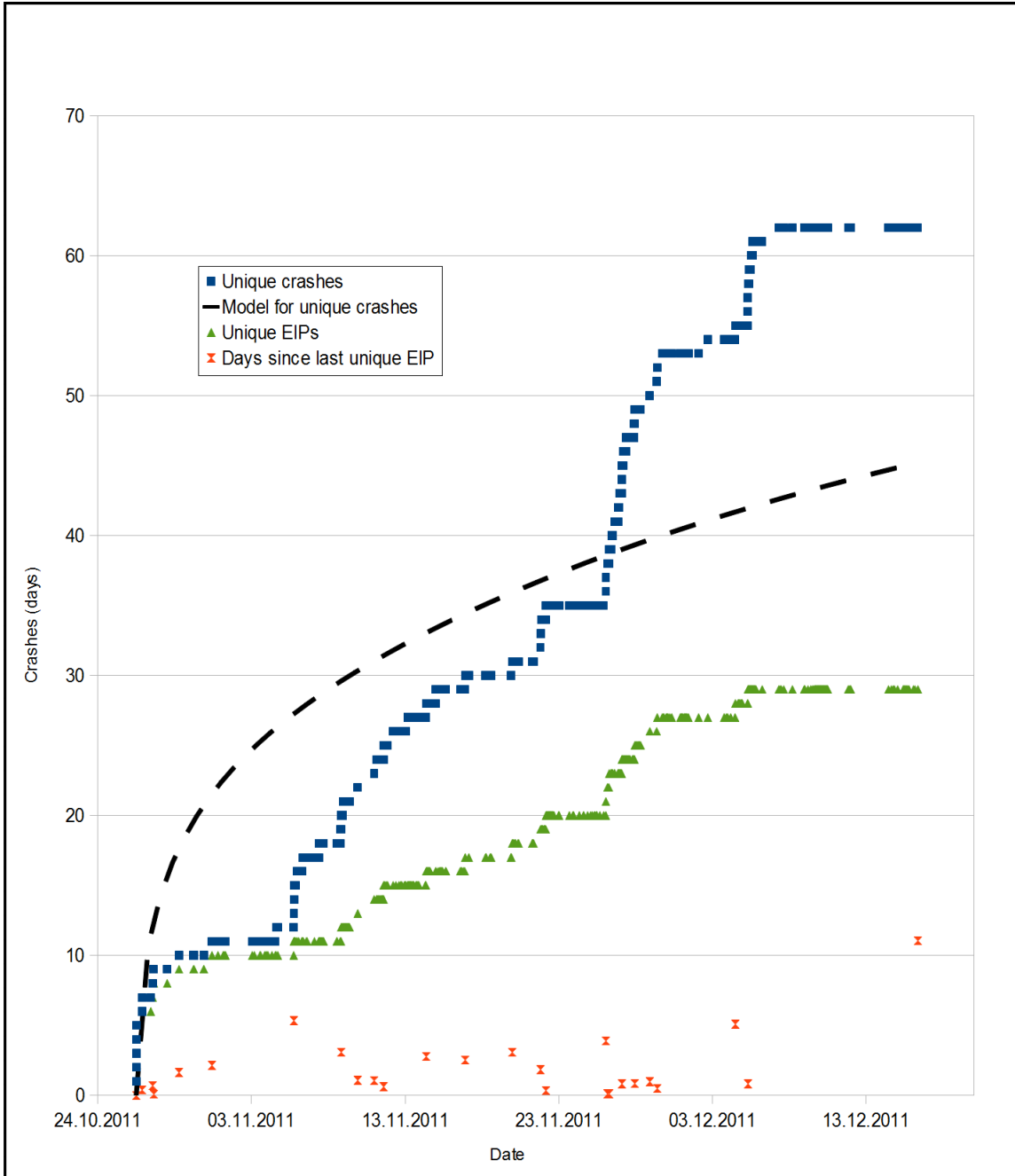


Figure 8.1: Progress of fuzz testing

was incremented. Similarly, it was identified when new EIPs were first detected, incrementing the counter of unique EIPs. These two counters were plotted in a graph shown in Figure 8.1. The graph also shows a theoretical curve for the unique crashes and the observed amount of days between unique EIPs. Time is used for the x-axis rather than iterations. There is a practical reason for this. The 8 VMs running Peach all had asynchronous iteration counters, but the time of each crash was logged.

The empirical curves show an almost linear increase of unique crashes and EIPs. Exceptions are in the beginning and the end of the curves. In the beginning the two curves quickly increase, and in the end they become horizontal. The first seven crashes were discovered by running the fuzzing templates. In fact 24 of the 10.000 PS files downloaded generated a crash without being mutated. These crashes had seven unique call stacks and six EIPs. The middle part of the curves show that new unique crashes were detected regularly throughout the fuzzing run.

Figure 8.2 shows how many times each unique crash was detected on a logarithmic scale. There are large variations among the crash groups. For example the four crash groups 001, 002, 004 and 028 contain 81.5% of all crashes while only representing 6.5% of the groups.

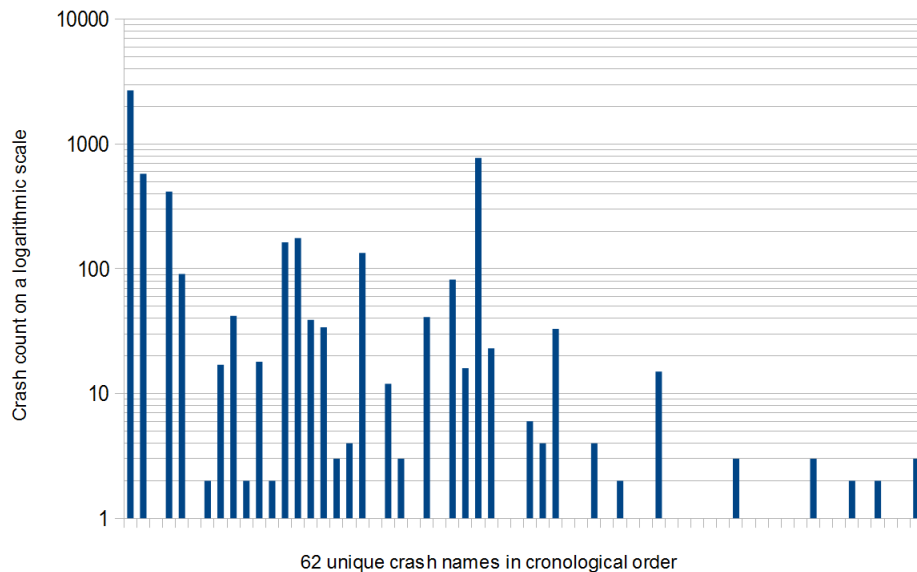


Figure 8.2: Frequency count of unique crashes

One interpretation of the frequency distribution is that most of the crash groups were rare during the fuzzing run. A trend can also be seen over time. An increasing part of new crashes were triggered only three times or less. During the second half of the fuzzing run, only rare crashes were added to the set of unique crashes.

The curve of unique crashes can be compared to the theoretical model described in Chapter 4 by using the observed frequency distribution. The theoretical curve shown in Figure 8.1 was produced by using  $k = 62$  and  $m = 5462$ . The frequency count for each unique crash was used as the

constants  $q_1$  to  $q_{62}$ . The value of  $Q$  affects the time horizon of the theoretical curve. It was adjusted manually to fit the empirical curve.

The empirical curve is more linear than the theoretical model. An explanation for this can be the distribution of program functionality among the fuzzing templates. Instead of picking a random mutation from a random template on each iteration, many iterations were run on a template before changing files. Each time a new template was picked, this would probably increase code coverage. This could be seen as a change of strategy. As described in Section 4.3 a change of strategy might raise the ceiling. When continuously exploring new parts of the program, new unique crashes were detected.

The last nine days of fuzzing did not give any additional unique crashes. However, as the graph shows, there were crashes, and the crashes collected in this period could add information to the existing crash groups. Since the crash counters were constant for a long period, it was decided to stop the fuzzing run. The data set was also assessed to be sufficiently diverse. The 62 crash names includes all four classifications and nine of the possible descriptions from the rules of !exploitable (Appendix B).

## 8.2 Results of the crash verification process

Crashes were verified by running one mutated input file from each group of unique crashes ten times. All crashes were verified, but none of the inputs produced a crash every time. This unreliability was unexpected in a single-threaded program given a single input file. Crash verification was also run against other versions of the program.

### 8.2.1 Analysis of reliability

The verification process showed a large variation in reliability. Some crashes were verified one out of ten times, others eight out of ten. A lesson from this, is that it is not sufficient to run a test case once more and see if the same crash occurs. Running the test several times can give an indication of the reliability of the crash. Also, a single input can give multiple crashes with different probabilities. This can only be discovered if the same input is run several times.

A probable cause for the unreliability is the use of uninitialized variables. Uninitialized data can have random values. If such random values are interpreted as memory addresses it could lead to an access violation on the instruction dereferencing the address. An access violation would only occur if the corrupted address points to an invalid memory region. If the address was corrupt but valid, the instruction would execute, leading only to a change in program behavior. It could lead to a different crash or the corruption could be properly handled and the program could exit with an error code.

During reliability testing it was also identified an inherent instability in the target program. This occurred when running Ghostscript on

Windows XP. Unfortunately this was the OS used during fuzzing, so crash verification also had to be performed on Windows XP.

The instability occurred even when a non-existing input file name was given as an argument. The correct behavior is a program termination with "exit code 1", but this happened only three out of four times. Approximately one out of four executions resulted in an early exit with the error code `gs_error.Fatal` (-100). This affected the measurement of reliability and made automatic verification difficult.

The randomness seemed to be related to system resources and OS memory management. After running the program several times, the error became more frequent. The random fatal error is not a crash, but it causes an abnormal termination with an error code. It is reproducible and independent of input. The error code is a symptom of a bug, and in that respect the situation is equivalent to a crash.

From the set of unique crash names 60 out of 62 crashes were reproduced on Windows XP with the same result as during the fuzz testing. The inputs for crashes 006 and 007 both produced the same result as 001. These two crashes were discovered and logged on Windows 7 on the initial run of all downloaded PS-files. They were reliably reproduced on Windows 7. Running all test cases on Windows 7 produced different results. Most crashes were verified, some did not produce a crash and others produced different crashes. All crashes on Windows 7 were reliable, i.e. either 0% or 100%.

### 8.2.2 Verification of test cases on other program versions

Since the verification process was automated, it was possible to run the same test cases on different versions of the program. This could test the assumption regarding the compilation of Ghostscript 6.51. It was assumed that the original executable files from 2001 would produce more crashes than a new compiled version.

The results showed a small reduction in unique crashes compared to the fuzz testing. 54 unique crashes were reproduced on 25 EIPs. Crashes 004 and 026 were not reproduced, and some of the different crashes produced the same result with the new program. Since the two programs were compiled with the same source code, the difference must lie in the compiler used. This comparison supports the assumption that the original was a better target for producing many different crashes.

Another natural task was to run the verification process on the newest version of Ghostscript. The test cases were run on version 9.04. This resulted in four unique call stacks and two new crash descriptions. The first one was `PROBABLY_EXPLOITABLE_TaintedDataControlsCodeFlow` produced by 054. The second was `EXPLOITABLE.StackCodeExecution` produced by 025, 043, 044 and 046.

The original five crash groups had ten equal stack frames counting from the top. The four new crashes also had similar call stacks. As expected, input files producing related crashes in the old program also gave related crashes in the new program. These relations are discussed in Chapter 10.

## Chapter 9

# Crash analysis

This chapter demonstrates crash dump analysis and dynamic analysis using the set of crashes from Chapter 8. The chapter is divided into three sections corresponding to the research questions of this thesis. Section 9.1 illustrates how relations between crashes can be identified by comparing call stacks (RQ1). Section 9.2 describes how crashes can be prioritized (RQ2). Section 9.3 is a case study of root cause analysis using different methods (RQ3).

### 9.1 Call stack analysis

This section compares different methods of grouping crashes based on their call stacks. Before grouping methods were analyzed, some call stacks needed to be reconstructed.

#### 9.1.1 Reconstruction of call stack

A call stack is an ordered list of functions that have been called but not yet returned. In the context of native program crashes, the call stack is a list of addresses. Each address points to the current program location in the called functions. The current location in the top stack frame equals the instruction pointer (EIP). For all other stack frames, it is the return address of the called functions, which is the saved EIP. If a return address is overwritten, the function responsible for that particular call will not show in the call stack. The top stack frame can also be missing if EIP points to an invalid address.

In the data set of crashes in `gsd1132.dll`, three unique crashes had corrupted call stacks:

- 023\_UNKNOWN\_TaintedDataPassedToFunction
  - corruption of all frames but the top stack frame (crash location)
- 045\_EXPLOITABLE\_WriteAV
  - corruption of all frames but the five top stack frames
- 052\_EXPLOITABLE\_ReadAVonIP
  - corruption of all stack frames, including the crash location

Execution traces were produced with TEMU using the input files that generated the three crashes. From each trace, a complete call graph was extracted as a chronological list of all function calls and returns. Figure 9.1 shows part of one of the call graphs and the corresponding call stack generated from this part only.

A call stack can be reconstructed from a call graph by simulating how the function stack works during execution. A stack data structure can be used to represent the function stack. For every function call in the call graph, the return address is pushed onto the stack. For every return, the top element is popped.

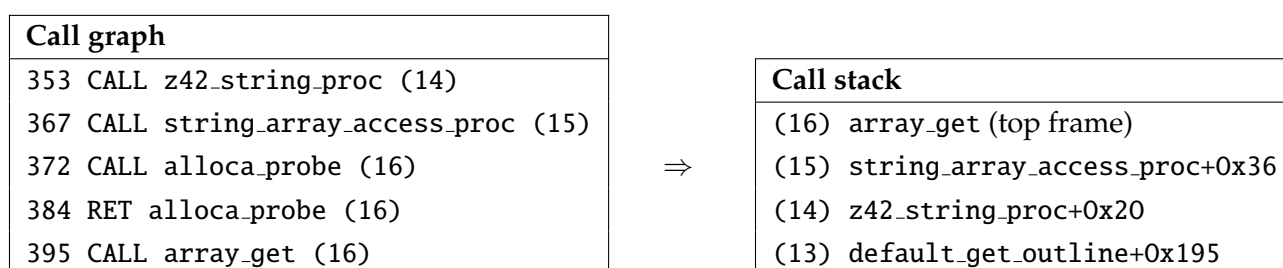


Figure 9.1: Call stack reconstruction

Parsing the call graph in Figure 9.1 gives a call stack with four stack frames. The top frame is the last function called. The other three frames are return addresses. The stack frame of `alloca_probe` is not shown because it has returned. The bottom frame shows a function name not present in the call graph. This is the function containing the first CALL instruction.

The numbers before the CALL and RET instructions are indexes into the execution trace used as the source of the call graph. For CALL instructions this index reveals the return address of the function call. The return addresses were resolved into function offsets by using the known start addresses of all functions. The numbers in parentheses show the current level of nested function calls, or *stack depth*.

The reconstructed call stacks of 023 and 045 were similar to the call stacks of other crashes. These relations were identified by using call stack based grouping algorithms. 052 did not group with any other crashes based on call stack, but Section 9.3 will show that it is in fact closely linked to 045.

### 9.1.2 Call stack based grouping algorithms

The purpose of crash grouping is to isolate crashes that are caused by the same bug. The data set of 62 unique crashes showed a grouping potential. For example 048 and 057 had a sequence of 17 equal stack frames counting from the top. When most bugs are fixed within the top ten stack frames [78], it is very likely that these two unique crashes are caused by the same bug.

Chapters 4 and 6 describe two algorithms for grouping crashes based on call stacks. One is a hash algorithm used by !exploitable. Another is a two-level approach using crash location for the first level and Levenshtein distance (LD) of call stacks for the second level. In addition to the described



algorithms, a simple comparison algorithm was created for this thesis. Variations of the algorithms were also tested.

The algorithm shown in Figure 9.2 performs a top-down sequential comparison of stack frames. The call stacks are grouped by always preferring call stacks with the longest chain of equal stack frames.

---

**Input:**  $cs \leftarrow$  list of call stacks,  $depth \leftarrow 0$ ,  $maxdepth \leftarrow$  integer  $\geq 1$

```

procedure TOPDOWN( $cs, depth$ )
  if  $cs = \emptyset$  then
    return  $\emptyset$ 
  if  $depth = maxdepth$  then
    return GROUP( $maxdepth - 1, cs$ )
  if  $len(cs) = 1$  then
    return GROUP( $depth, cs$ )
   $groups \leftarrow \emptyset$ 
   $unique \leftarrow$  list of unique frames at current depth
  for all  $f \in unique$  do
     $equals \leftarrow$  list of elements from  $cs$  with  $frame = f$  at current depth
     $subgroups \leftarrow$  TOPDOWN( $equals, depth + 1$ )
    if LENGTH( $subgroups$ ) = LENGTH( $equals$ ) then
       $groups \leftarrow groups +$  GROUP( $depth, equals$ )
    else
       $groups \leftarrow groups + subgroups$ 
  return  $groups$ 

```

**Output:** List of group elements of the form  $(d, cs)$  where  $cs$  is a list of grouped call stacks and  $d$  is the depth at which the grouping was performed.

---

Figure 9.2: Top-down comparison algorithm for call stack grouping

The two-level approach of the LD algorithm is proposed mainly to minimize the computation needed to do the grouping. A possible negative side-effect is that different crash locations can separate crashes with otherwise identical call stacks. If it is feasible for a given number of crashes, a one-level approach might create a grouping that corresponds more to the actual bugs in the program.

An open question is if function offsets should be included in the comparison of stack frames. If there are large functions, omitting the offsets might cause grouping of unrelated crashes. On the other hand, if stack frames with equal functions and different offsets are treated as unequal, it might cause separation of related crashes.

To answer what effect the choice of grouping algorithm might have, several algorithms were tested. The LD algorithm was implemented with and without function offset. A one-level LD algorithm was also tested, treating all crashes as one crash-type. The LD algorithms were implemented with incremental grouping and representative traces.

### 9.1.3 Comparison of grouping algorithms

Table 9.1 shows a comparison of the tested grouping algorithms. Levenshtein distances were calculated between all unique crashes in a group. Trace diversity and Silhouette values were calculated for all groups containing more than one unique crash. Hence  $TD_{avg}$  and  $S_{avg}$  are not affected by the single groups.

Algorithm	groups	> 1	largest	$LD_{max}$	$TD_{max}$	$TD_{avg}$	$S_{min}$	$S_{avg}$
0 Unique names	1	1	62	10	8.37	8.37	N/A	N/A
1 !exploitable minor hash	61	1	2	0	0.00	0.00	1.00	1.00
2 !exploitable major hash	26	10	16	4	4.00	1.17	0.00	0.77
3 Crash function+offset	29	11	19	9	7.00	2.89	-1.00	-0.28
4 Top-down compare offset	36	16	6	7	7.00	1.62	-1.00	0.19
5 Two-level LD offset	38	10	10	1	1.00	0.15	-1.00	0.03
6 One-level LD offset	30	10	16	1	0.53	0.05	0.84	0.94
7 Crash function	21	9	27	9	6.00	2.85	-0.43	0.39
8 Top-down compare function	29	13	9	6	6.00	0.77	0.33	0.95
9 Two-level LD function	30	10	16	1	0.53	0.05	0.84	0.94
10 One-level LD function	28	11	16	2	2.00	0.32	0.00	0.88

Table 9.1: Comparison of grouping algorithms

Table 9.1 shows that the number of groups should lie somewhere between 26 and 30. Five of the algorithms (2, 6, 8, 9 and 10) produce a grouping within this range while keeping  $TD_{avg}$  low and  $S_{avg}$  high. Two of these algorithms (2 and 10) give an  $S_{min}$  of zero. This indicates that at least one crash in these groups should be separated into its own group. Section 10.1 discusses Table 9.1 in more detail.

### 9.1.4 Crash graph analysis

The tested grouping algorithms compare the top stack frames in order to find relations between call stacks. These relations can be visualized by using a *Crash Graph* [50] as described by Kim et al. Figure 9.3 shows the graph generated from all stack frames within `gswin32c.exe` and `gsdl132.dll`. It was created using Gephi [3], an open source software for graph visualization and manipulation.

This crash graph is a directed graph containing 179 nodes and 205 edges. The nodes are stack frames, and the edges are function calls. First all unique stack frames were identified, comparing both functions and offsets. Then each unique call stack was traversed bottom-up, generating directed edges between stack frames.

The edges are weighted by the occurrence of a particular call relation in the set of call stacks. Similarly, nodes are weighted by the occurrence of a particular stack frame. In addition, a linearly increasing weight is added to the ten nodes closest to crash nodes. There are 29 crash nodes shown in

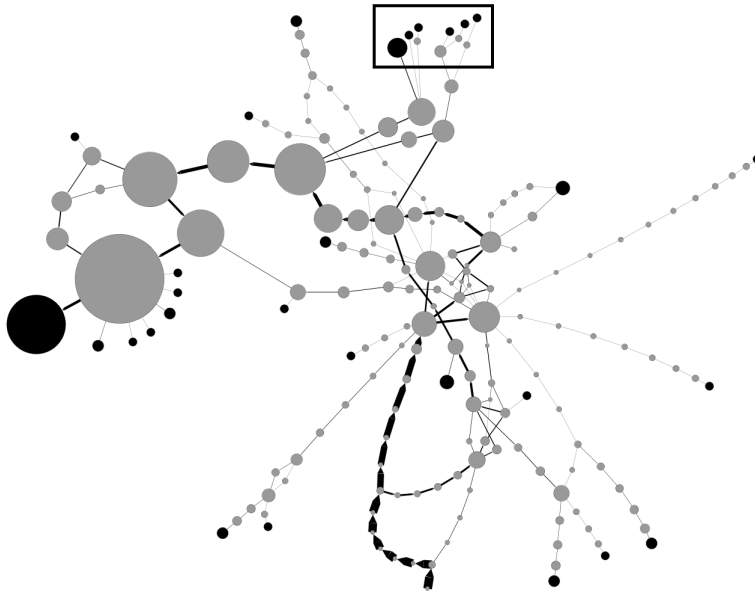


Figure 9.3: Crash graph of all stack frames

black. They are identified as the top stack frames and correspond to the EIP groups.

The topology of the graph shows some characteristics of the data set. Starting from the bottom there is a sequence of thick edges. This path is shared by most of the call stacks, and is probably irrelevant to the crashes. Some crash nodes are close, while others are more isolated. For example the crash node to the upper right is in the end of a sequence of 11 stack frames not shared with any other call stack. The largest node of the graph, to the left, is the stack frame of `check_component`. This stack frame is shared by 27 different call stacks resulting in seven different crashes in `append_simple`, represented by black nodes.

Six crash nodes at the top of the graph are shown in more detail in Figure 10.3. These are all crashes containing stack frames from the function `gs_type1_interpret`. Section 10.1 discusses how a crash graph can be used to identify and understand relations between crashes.

## 9.2 Prioritizing crashes

The crashes which are most frequent in field crash reports should be given a high priority. Removing the most common crashes first will improve stability of the program and fix problems for a large segment of users. This strategy is recommended for developers using WER [39].

It may be assumed that this is the case also for crashes produced by fuzzing. A crash that can be triggered by many different inputs is more liable to occur in ordinary use than a crash that is rare during fuzz testing. On the other hand, a rare crash can be classified as more severe, giving it a higher priority for fixing. Reliability is another aspect related to frequency. A crash occurring reliably during testing might affect more users than an unreliable crash.

This section presents calculated metrics for the crashes of this thesis. Then taint analysis is demonstrated as a method for assessing the possible security impact of a crash. This can be a complementing method to automatic classification based on crash dump analysis.

### 9.2.1 Comparison of prioritization metrics

Figure 8.2 showed a large variation in frequency among the unique crashes. Chapter 8 also discussed a large variation in reliability. Table 8.1 showed severity estimates for the crashes. These three metrics can all be used to prioritize the crashes for bug fixing.

Table 9.2 shows a comparison of metrics for all 29 crash locations. The table is sorted by frequency. Since some crash locations are much more frequent than others, starting from the top of the list might remove problems for most users first. For example removing the cause of 20% of the crash locations could remove 95% of the crashes. This would mean analyzing and fixing only the top six crash locations from the sorted list.

The bottom part of the table shows the most rare crashes. Among the most rare crashes are also the most severe classifications. These classifications can be used to give severe crashes a higher priority. For instance crashes with a classification of `EXPLOITABLE` or `PROBABLY_EXPLOITABLE` could be fixed before all others. This is particularly important if the program receives input from the Internet, making it a possible attack vector for malware.

The fifth column of Table 9.2 shows the maximum reliability of crashes for each crash location. This is the average verification rate of the most reliable crash within the group. All 5462 input files were tested in order to find the most reliable inputs. Because of the inherent instability discussed in Section 8.2, no crash is given a higher reliability than 75%.

It would be easier to use this column for prioritization if some crashes were 100% reliable. The most reliable crashes should be given the highest priority, but if the accuracy of the measurement is low, this might not serve its purpose. However, there seems to be a relationship between frequency and reliability. The more frequent crashes are more likely to be reproduced.

### 9.2.2 Taint analysis

Execution traces were produced for the crashes that needed call stack reconstruction. The size of the traces varied from about 15 to 100 GB, and it took several hours to produce each trace. The discussed unreliability in the target program combined with available time and disk storage made automatic tracing of all crashes impractical. Consequently only a few example traces were produced.

Taint analysis was performed by marking the whole content of the input files as tainted. The tainted data was propagated through the system by TEMU. In the execution traces, taint information was logged for each instruction operand. If an operand is marked as tainted, it means that its value is influenced by tainted data.

Table 9.2: Comparison of prioritization metrics

Crash location	source	frequency	stacks	max rel.*	max exp.†
ref_param_make_int+0xe	iparam.c	2687	1	75%	2
igc_reloc_struct_ptr+0x7f	igc.c	775	1	45%	2
append_simple+0xa8	gstype42.c	630	19	75%	2
s_filter_close+0x80	stream.c	581	2	75%	2
pdf_put_colored_pattern+0x24	gdevpdfv.c	417	2	75%	1
s_DCTD_process+0x1f	sdctd.c	103	1	75%	2
pdf_font_notify_proc+0x8b	gdevpdfw.c	101	3	50%	1
pdf_write_embedded_font+0x143	gdevpdfc.c	83	2	75%	3
gs_text_replaced_width+0x99	gstext.c	43	3	75%	2
gs_type1_interpret+0x249	gstype1.c	7	5	60%	1
cos_dict_elements_write+0xa	gdevpdfo.c	6	1	60%	2
append_simple+0x43b	gstype42.c	4	1	50%	2
igc_reloc_struct_ptr+0x57	igc.c	3	2	0%†	2
append_simple+0x408	gstype42.c	3	2	60%	2
restore_finalize+0x81	isave.c	3	1	40%	2
gx_path_add_line_notes+0x12	gxpath.c	2	1	75%	2
append_simple+0x58	gstype42.c	2	2	50%	2
refset_null_new+0x29	iutil.c	1	1	60%	4
append_outline+0x10d	gstype42.c	1	1	60%	4
append_simple+0x538	gstype42.c	1	1	40%	4
gx_path_add_curve_notes+0x195	gxtype1.c	1	1	30%	3
gs_type1_endchar+0x18b	gxtype1.c	1	1	70%	2
append_simple+0x34e	gstype42.c	1	1	60%	2
append_simple+0x480	gstype42.c	1	1	50%	2
gs_type1_interpret+0xbae	gstype1.c	1	1	45%	2
append_component+0xe4	gstype42.c	1	1	40%	2
s_zlib_free+0x93	szlib.c	1	1	40%	2
pixel_resize+0x55	gdevpsdi.c	1	1	35%	2
type1_apply_path_hints+0x1c	gxhint3.c	1	1	75%	1

\* max reliability

† This crash location is not reproduced on Windows XP, only on Windows 7

‡ max exploitability

1 = PROBABLY\_NOT\_EXPLOITABLE

2 = UNKNOWN

3 = PROBABLY\_EXPLOITABLE

4 = EXPLOITABLE

In general, if the operands of a failing instruction are tainted, it increases the chance that the crash can be controlled and exploited. This is in particular true for write access violations. If an invalid write operation can be manipulated into overwriting an important data structure, it might be used to hijack control flow [70].

Figure 9.4 shows the result of taint propagation for three execution traces. Read and write operations are marked by R and W. Untainted values are marked by T0 and tainted values are marked by T1. CPU registers are marked by R@, and memory locations are marked by M@. The taint information shows a pair of integers for each tainted byte. The first is an identifier for the tainted data, and the second is an offset into the data.<sup>1</sup>

---

#### 052\_EXPLOITABLE\_ReadAVonIP

```

mov esp,ebp    R@ebp[0x0006ebac][4](R) T0 R@esp[0x0006e6e4][4](W) T0
pop ebp       M@0x0006ebac[0x04e8dd98][4](R)
              T1 {15 (634, 778290) (634, 778290) (634, 778290) (634, 778290)}
              R@ebp[0x0006ebac][4](W) T0
ret           M@0x0006ebb0[0xffa961a8][4](R)
              T1 {15 (634, 778290) (634, 778290) (634, 778290) (634, 778290)}

```

#### 023\_UNKNOWN\_TaintedDataPassedToFunction

```

mov ecx,[ebp+8] M@0x0006bb68[0x00025000][4](R)
               T1 {15 (980, 507939) (980, 507939) (980, 507939) (980, 507939)}
               R@ecx[0x00000003][4](W)
               T1 {15 (980, 507939) (980, 507939) (980, 507939) (980, 507939)}
mov edx,[ecx+8] M@0x00025008[0x00000000][4](R) T0
               R@edx[0x0000ab92][4](W)
               T1 {15 (980, 507939) (980, 507939) (980, 507939) (980, 507939)}

```

#### 003\_PROBABLY\_NOT\_EXPLOITABLE\_ReadAVNearNull

```

mov edx,[ebp+0x10] M@0x0006c5dc[0x00000000][4](R) T0 R@edx[0x00000000][4](W) T0
mov eax,[edx+0x1c] M@0x0000001c[0x00000000][4](R) T0 R@eax[0x0006ca74][4](W) T0

```

---

Figure 9.4: Taint information from three crashes

The first trace shows two tainted values being read from the stack. The first value is the restored frame pointer, and the second value is the return address. The fact that all four bytes of these values are marked as tainted, strengthens the assumption that this crash may be controlled by user input. This may allow malware to take control of the program and possibly compromise the system.

The second trace shows the dereference of an invalid address, 0x25008. The address is calculated as an offset of eight from a tainted value (ECX). This is a situation described by Miller et al. to be uncertain [68]. An invalid read not near null might pass if memory were allocated on this

---

<sup>1</sup> 15 = 0x1111 is a bitmap showing that all four bytes are tainted. Only the first taint of a byte is logged, even if it is actually influenced by multiple sources.

address. The data at this location would then be passed as a function argument. Depending on the called function this may or may not result in an exploitable situation.

The called function is `type1.apply_path_hints` which may be given an invalid pointer `ppath` as an argument. In fact another unique crash occurred at the beginning of this function while dereferencing `ppath`. The last trace shows this crash, which is a null pointer dereference classified as probably not exploitable. Taint analysis supports this classification because the null pointer (EDX) is not tainted.

## 9.3 Root cause analysis

Root cause analysis was performed on one of the unique crashes. The name of the crash is `052_EXPLOITABLE_ReadAVonIP`. Crash dump analysis was used to inspect the program state at the moment of crash. An execution trace was used to inspect historic program states, leading to the crash.

### 9.3.1 Crash dump analysis

A crash dump usually reveals the crash location and a call stack showing previous stack frames. However, automatic analysis of crash 052 revealed no stack frames, and the crash location was invalid. An unhandled exception occurred because the instruction pointer (EIP) contained an invalid address, and the last valid EIP is not present in the crash dump. This is similar to the scenario showed in Figure 3.11. Dynamic methods can be applied to reconstruct the call stack and the crash location. It might also be necessary to use dynamic analysis to understand the root cause, but the crash dump might contain valuable information.

Inspection of the stack memory in the crash dump showed a distinct data pattern at the top of the stack, given by the stack pointer (ESP). The pattern was a continuous array of elements of eight bytes. The array went far down the stack, indicating a stack buffer overflow. The values of EIP and the frame pointer (EBP) could be found at `[ESP-4]` and `[ESP-8]`. This indicates that these stack locations contain the return address and saved frame pointer of a stack frame.

As showed in Figure 3.8, the saved frame pointers form a singly linked list. Each saved frame pointer points to the stack location of the frame pointer from the previous stack frame. The return address is stored directly below the frame pointer. This can be used to identify stack frames of returned functions.

A minidump does not contain stack memory of returned functions. However, a complete dump of the stack memory region revealed six return addresses above the overflow pattern, on lower addresses. The frames were identified by performing a backward traversal of the singly linked list of frame pointers. This was done by manually searching for the addresses of stack locations containing frame pointers. The result is shown in Table 9.3. The revealed stack frames correspond to the last functions of the call graph, shown in Table 9.4.

Address	Value	Return address → Called function
0006e584	<b>0006e5b0</b>	
0006e588	10020386	gsdll32!string_array_access_proc+0x36 → array_get
0006e5b0	<b>0006e5cc</b>	
0006e5b4	10020741	gsdll32!z42_string_proc+0x20 → string_array_access_proc
0006e5cc	<b>0006e5f8</b>	
0006e5d0	10020ff9	gsdll32!default_get_outline+0x195 → z42_string_proc
0006e5f8	<b>0006e658</b>	
0006e5fc	10021e34	gsdll32!total_points+0x22 → default_get_outline
0006e658	<b>0006e6c4</b>	
0006e65c	10022a76	gsdll32!append_component0x169 → total_points
0006e6c4	<b>0006ebac</b>	
0006e6c8	10021d7a	gsdll32!append_outline+0x76 → append_component
...		
0006eba8	ffa9b194	
<b>0006ebac</b>	04e8dd98	([ESP-8] = EBP = 0x04e8dd98)
<b>0006ebb0</b>	ffa961a8	([ESP-4] = EIP = 0xffa961a8)
0006ebb4	04e76f80	(ESP = 0x0006ebb4)
0006ebb8	ffa953e8	
...		
0006fd1c	<b>0006fd34</b>	
0006fd20	10007264	gsdll32!gsdll_init+0x94 → gs_main_init_with_args
0006fd34	<b>0006fd54</b>	
0006fd38	00401c16	gswin32c!gsdll_class::init+0xdd → gsdll_init
0006fd54	<b>0006ff70</b>	
0006fd58	00401182	gswin32c!new_main+0xb2 → gsdll_class::init
0006ff70	<b>0006ff80</b>	
0006ff74	00401222	gswin32c!main+0x1d → new_main
0006ff80	<b>0006ffc0</b>	
0006ff84	00402e91	gswin32c!start+0xc5 → main
0006ffc0	<b>0006fff0</b>	
0006ffc4	7c816d4f	kernel32!RegisterWaitForInputIdle+0x49

Table 9.3: Recovered stack frames from a stack memory dump



Table 9.3 also shows return addresses from below the overflow pattern. These could not be found by following frame pointers, because the linked list was disrupted by the overflow. They were identified by analyzing addresses stored on the stack. Addresses that were in the address range of executable modules, were resolved into function offsets.

A return address can be identified as referencing a program location immediately following a function call. This can be used to avoid false positives. For example a function pointer references the beginning of a function, and can therefore not be a return address. When the first return address was found, the rest of the existing stack frames were identified by traversing the list of saved frame pointers.

This example shows that manual inspection of a crash dump might reconstruct parts of a corrupted call stack. Also a complete stack dump might contain the stack frames of returned function calls. In this particular case, it indicates that the stack frame of `append_outline` was overflowed. This could be found by inspecting only the program memory at the moment of crash. The source code in Figure 9.5 shows that `append_outline` contains a stack buffer named `pts` (line 879).

Identifying an overflowed buffer might be enough information to fix the bug. However, to properly understand the root cause, more questions should be asked. One question is where in the program the overflow took place. A second question is why the overflow was allowed to happen. These two questions cannot be properly answered by analyzing the crash dump, because it contains no reference to the responsible function.

Last part of the call graph for 052_ReadAVonIP	
438	RET array_get (16)
468	RET string_array_access_proc (15)
471	RET z42_string_proc (14)
482	RET default_get_outline (13)
536	RET total_points (12)
548	RET append_component (11)
553	RET spurious (0) (Invalid return address)

Table 9.4: Continuation of the call graph from Figure 9.1

### 9.3.2 Analysis of an execution trace

An execution trace of crash 052 was produced, and a complete call graph was extracted from the trace. The last part of this call graph is shown in Table 9.4. The last instructions of the execution trace are shown in Figure 9.4. The addresses of these instructions confirmed that the crash occurred at the return of `append_outline`. The call stack was reconstructed from the call graph. This revealed 15 additional return addresses that were overwritten in the crash dump.

To answer where the overflow took place, a search was performed

backward in the execution trace for the address 0x0006ebb0. This is the stack location of the corrupted return address, as shown in Table 9.3. The approach is similar yet simpler than the methods shown in case studies by Miller et al.[68] They used slicing, aligning and allocation tracking to isolate important code. The approach used here performs only a simple text search for references to an important memory location.

The last reference to the stack location was a write operation in the function `append_simple`. In fact, the overwrite occurred in the same location as the write access violation of crash 045. Call stack reconstruction at this point in the execution trace showed that the top five stack frames were equal to the call stack of 045. The last of these five stack frames was also the last valid stack frame from the crash dump of 045. It was the return address from `append_component` into `append_outline`. This close relationship could not be found without dynamic analysis of 052, because it required information about program states prior to the crash.

To understand how a write operation was able to write past the end of a buffer, consider the source code shown in Figure 9.5. The overflowed buffer is `pts` (line 879) defined in `append_outline`. If the variable `num_points` is 150 or less, the stack buffer is used. If it is greater than 150, a buffer is dynamically allocated on the heap, not shown in this code listing. The buffer pointer is passed as an argument to subsequent function calls.

Furthermore, the function `append_component` may add the value of `point_index` to the buffer pointer (line 808). The index variable starts with the value zero (line 881) and may be incremented in a loop (line 850). This gives `check_component` an argument `ppts` that may point beyond the start of the buffer. This argument is passed to `append_simple` (line 791) which copies point values with an element size of eight bytes into the buffer (line 685). The length of the copy is limited by the variables `numContours` and `last_point` which are both calculated from `gdata` (lines 582-649).

The execution trace can help understand what went wrong by showing the values of certain variables. The values can be inspected by searching backward for program locations where the respective variables are used. In this case, the search was performed from the point in the trace where the overflow occurred and backward. The results of the search is shown in Table 9.5.

Variable	Value
<code>num_points</code>	63
<code>point_index</code>	60
<code>numContours</code>	2
<code>last_point</code>	513

Table 9.5: Variable inspection of 052\_EXPLOITABLE\_ReadAVonIP

Crash dump analysis of 045 showed that it created an access violation during the overflow when trying to write past the bottom of the stack. A value inspection of `last_point` in the execution trace of 045 showed that

```

579 // append_simple
...
582 int numContours = S16(gdata);
583 const byte *pends = gdata + 10;
...
638 for (i = 0, np = 0; i < numContours; ++i) {
...
642 uint last_point = U16(pends + i * 2);
...
649 for (; np <= last_point; --reps, ++np) {
...
682 pt.x += dpt.x, pt.y += dpt.y;
683
684 if (ppts) /* return the points */
685     ppts[np] = pt;
// 052: overflow here. 045: overflow resulting in Write AV
...
772 //check_component
...
782 code = pfont->data.get_outline(pfont, glyph_index,
                                &glyph_string);
...
785 gdata = glyph_string.data;
...
791 code = append_simple(gdata, sbw, pmat, ppath, ppts, pfont);
...
800 // append_component
...
807 code = check_component(glyph_index, pmat, ppath, pfont,
808                        ppts + point_index, &gdata);
...
820 do {
...
846 code = append_component(comp_index, &mat, ppath, ppts,
847                          point_index, pfont);
...
850 point_index += total_points(pfont, comp_index);
851 }
852 while (flags & cg_moreComponents);
...
857 // append_outline
...
875 #define MAX_STACK_PTS 150 /* usually enough */
876 int num_points = total_points(pfont, glyph_index);
877
878 if (num_points <= MAX_STACK_PTS) {
879     gs_fixed_point pts[MAX_STACK_PTS];
880
881     return append_component(glyph_index, pmat, ppath, pts, 0,
882                            pfont);
// 052: Read AV on EIP at the return from append_outline

```

Figure 9.5: Excerpts from gstype42.c

it had the large value 3331. In the case of 052, all write operations of the overflow were legal. The upper limit `last_point` was large enough to overflow the buffer, but small enough to avoid writing past the stack. The crash did not occur until an overwritten return address was used.

Figure 9.6 shows the layout of the buffer `pts` and how different variables affect the overflow. If `num_points` were greater than 150, the buffer would be allocated on the heap. This might result in a heap overflow starting at `num_points`.

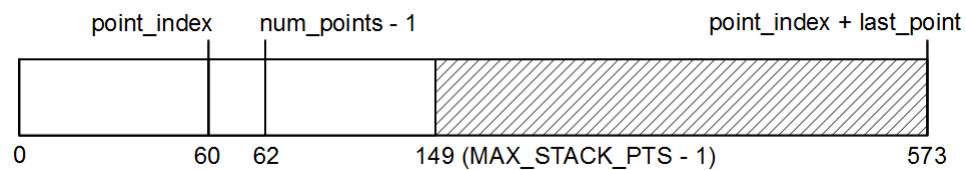


Figure 9.6: Overflow of the buffer `pts` caused by `append_simple`

The overflow could be avoided by passing `num_points` as an argument to `append_component` (line 881) giving the number of points in the buffer. This value should be updated by subtracting `point_index` giving the amount of points left as an argument to `check_component` and `append_simple`. This argument could be used to calculate an upper boundary for `last_point`.

If `last_point` exceeds the upper boundary, the program should perform suitable error handling. An alternative is to dynamically increase the buffer size if the buffer is found too small to contain the point array. This cannot be done for a stack buffer, which is statically allocated, but a heap buffer may be reallocated to a new size at runtime.

### 9.3.3 Input analysis

Operand values from the execution trace of 052 showed that the stack overflow was caused by improper boundary checking of the variable `last_point`. A still unanswered question is why this variable became so large in the first place.

The value comes from `gdata = glyph_string.data` (line 785). Comparing the mutated file with the original showed a mutation inside a Type42 [9] font description embedded in the PostScript file. A Type42 font is a TrueType [27] font encapsulated in PostScript format. The mutation causing crash 052 is shown in Appendix A.

It is likely that the mutated font descriptions generating crashes 045 and 052 are invalid with respect to the data format of font descriptions. If so, the source code responsible for reading the font descriptions from input should also be reviewed. Proper input validation may enable error handling at an early stage, so that invalid font descriptions are not used by the program.

**Part V**

**Discussion**



## Chapter 10

# Answering questions about program crashes

This chapter discusses how the methods demonstrated in Chapter 9 can help answering the three research questions of this thesis. The methods are discussed in the context of fixing bugs. This discussion is based on both existing research and the experiences made by exploring current methods.

### 10.1 RQ1: How are crashes related?

This section discusses different strategies of finding relations between crashes. Ideally, one group should be created for each bug, and the crashes should be grouped accordingly [39]. However, the actual relations between crashes and bugs are not yet known at the time of grouping. Grouping strategies use a set of *condensing* and *expanding* rules to group and separate crashes with the goal of reducing the time needed to fix each individual bug.

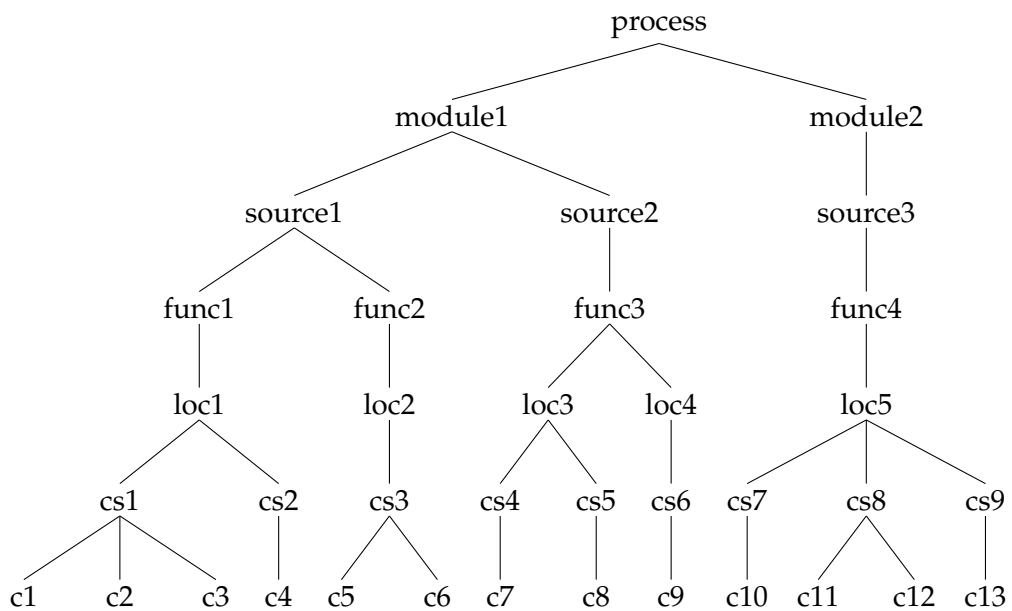
#### 10.1.1 Expanding rules

Expanding rules are used to separate crashes that may be caused by different bugs. An example is the two-level algorithm using Levenshtein distances [33]. First crashes are separated into crash-types based on their crash locations. Then crashes within a crash-type are separated into subgroups by comparing call stacks. Table 9.2 shows a potential of extending this approach to more levels.

Some crashes occurred at different offsets within a function. The grouping algorithms tested in Section 9.1 either used the function name or the offsets in functions to compare stack frames. A different approach could be to do both. Crashes could be grouped by function on one level and by function offset on the next level. It could also be possible to group crashes by source file. This could be a level between module and function. A program may also contain more than one process, which means that process name can be used as a top level.

Field crashes and crashes generated by fuzz testing of a program could first be separated into groups based on the crashing process. Grouping at the next levels could be decided by the crashing module, source file and function. Crashes could be further separated based on the crash location inside functions. A final expanding rule could separate crashes with different call stacks.

Figure 10.1 shows a generic example of such an expanding sort tree. When moving down the tree, the number of groups is increasing, and the amount of crashes in the groups is decreasing. The nodes at the bottom level contain individual crashes. In such a tree structure, each node is a superset of its children.



process  $\supseteq$  module  $\supseteq$  source  $\supseteq$  function  $\supseteq$  location  $\supseteq$  call stack  $\supseteq$  crash

Figure 10.1: Expanding sort tree

It will depend on the program how many levels that are suitable for an expanding sort tree. Also there must be a one-to-many relationship between the levels. For example, one source file may contain several functions, and one function may contain several crash locations.

An advantage of the expanding sort tree is that a bucketing algorithm can be used for grouping crashes. Such an algorithm sorts each new crash in a deterministic manner. It does not matter in which order new crashes arrive to the grouping algorithm. Also new crashes can be grouped without comparing to the existing crashes.

When unique call stacks are identified by !exploitable, a hash algorithm is used. The hash can be used for labeling buckets. The call stack level of the sort tree could be extended by using both the major and minor hash algorithms of !exploitable. Crashes could first be grouped by major hash. Then the groups could be expanded by the minor hash.



### 10.1.2 Condensing rules

Condensing rules are used to group crashes that may be caused by the same bug. As shown in Section 9.1 the crash location and call stack can be used to find relations between crashes.

The sort tree shown in Figure 10.1 is strictly expanding. However, source files might be shared between modules. Different modules might also use the same functions from these source files. This gives a potential for grouping crashes across different modules, if the crashing functions are equal. Using the top stack frame to perform grouping can therefore be seen as a condensing rule with respect to the expanding sort tree.

Call stack comparison and crash graph analysis might identify similarities further down the call stacks. This can motivate grouping of crashes with different crash locations. This could be seen as a further condensing rule, reducing the amount of groups.

### 10.1.3 Call stack based grouping algorithms

Call stack grouping is used to identify crashes that are caused by the same bug. Call stacks can be used for this purpose, because they contain program paths leading toward crashes [78, 33]. The method is effective because it can compare control flow by using crash dumps only. An exception is if the call stack is corrupted. This is normally caused by stack buffer overflows, and these crashes should anyway be given special attention because of their possible security implications [15, 70].

Table 9.1 shows a comparison of different call stack based grouping algorithms. The algorithms are evaluated by comparing trace diversity and Silhouette values. The best algorithms minimize the number of groups while keeping trace diversity low and Silhouette values high. Algorithms 6 and 9 produce the best groupings by these criteria. The two algorithms both produce the same set of 30 groups. These are listed in Appendix E.

No analysis was done to assess whether the 30 call stack groups really correspond to 30 different bugs. Instead, the evaluation is based on the results of Dhaliwal et al. [33] A key element is that it is better to separate into more groups than to group crashes from different bugs. If two call stacks are significantly different, the crashes should be analyzed separately, even if they are caused by the same bug.

To understand how the call stack groups can affect bug fixing time, they can be compared to the 29 EIP groups (algorithm 3). The largest Levenshtein distance ( $LD_{max}$ ) within the call stack groups is one, while it is nine for the EIP groups. This gives a  $TD_{avg}$  of 0.05 for the call stack groups and 2.89 for the EIP groups. This means that the crashes within the call stack groups are much more similar than the crashes within the EIP groups.  $S_{avg}$  is 0.94 for the call stack groups and -0.28 for the EIP groups. A negative Silhouette value for a given crash means that it is more similar to crashes in another group than to the crashes within its own group.

The negative value of  $S_{avg}$  indicates that a large portion of the 29 EIP groups contain crashes that belong to different bugs. By using call stack

comparison it was possible to produce a set of 30 groups with similar call stacks. The crashes in these groups are more likely to be caused by the same bug. Since the amount of groups is almost the same, it is probable that the total amount of time needed to locate and fix the bugs will be reduced. According to Dhaliwal et al. this approach could reduce bug fixing time in a set of field crash reports by more than 5%.

#### 10.1.4 Variations of grouping algorithms

Testing of different grouping algorithms showed that small adjustments to the algorithms could have a large impact on grouping. This supports the assumption that grouping algorithms should be tuned to fit different sets of crashes, for example by adjusting the grouping threshold [33].

The LD algorithms (5, 6, 9 and 10) use a threshold value for when new groups should be created. The stack trace of new crashes are compared to representative traces of the existing subgroups of the crash-type. A new subgroup is created if the LD between the new stack trace and each representative trace is greater than the threshold.

The use of representative traces showed to be necessary in order to keep TD below the threshold. The two-level LD algorithm was first implemented by comparing new call stacks to the first call stack in a group. This gave an  $LD_{max}$  of eight in the largest group when using a threshold of five. All call stacks had a distance of five or less to the first call stack, but their differences went in many directions. This gave a high TD for the group.

A threshold of five was suitable for the algorithm developed by Dhaliwal et al. [33] when grouping field crash reports. However, such a high threshold was not ideal for the fuzzing data set of this thesis. Fuzz testing of one specific program functionality might produce more similar crashes than the user generated crashes of a field crash reporting system. In any case, it is suggested that the threshold value should be adjusted to fit the properties of different programs.

A high threshold gives a high TD within groups. This increases the chance that a crash is more similar to other groups than its own. The threshold could therefore be tuned to find an ideal grouping. A threshold of two was chosen for the LD algorithms, giving a low TD and a high Silhouette value. The Silhouette validation used in this thesis compared crashes to all other groups, not only the groups within its crash-type. This factor may have contributed to why the threshold had to be lowered to produce high Silhouette values.

For this data set, the use of function offsets did not produce good results. Algorithms 3, 4 and 5 produced more groups with higher  $TD_{avg}$  and lower  $S_{avg}$ , compared to algorithms 7, 8 and 9. An exception is algorithm 6 which produced the exact same grouping as algorithm 9. What separates algorithm 6 from the three others (3, 4 and 5) is that it does not require the top stack frame to be equal within a group. It treats all crashes as belonging to the same crash-type. This algorithm performs significantly better than the three others. This is a strong indication that

the bias created by an initial level of grouping can have a negative effect if it is too expanding.

One could argue that this comparison is unfavorable for the algorithms using offsets. The comparison shown in Table 9.1 uses only function names to calculate LD, which is used for Silhouette validation. However, similar results are produced by using function name and offset for LD. Algorithms 3, 4 and 5 are given a lower  $S_{avg}$  than 7, 8 and 9 respectively, and these values are significantly lower than the  $S_{avg}$  for algorithm 6.

Table 9.1 shows that three of the LD based grouping algorithms produced good results (6, 9 and 10). This is as expected when the evaluation is based on LD. An interesting observation is that the major hash grouping (algorithm 2) and top-down compare of functions (algorithm 8) produced competitive results. Both of these use only function name to compare stack frames, and they group crashes with N equal stack frames counting from the top. For the !exploitable major hash, N is a fixed value of five.

These algorithms have in common that they do not allow any replacement or reordering of stack frames. Only sequentially equal stack frames are counted. Still they produce a relatively small amount of groups. Another aspect is that these algorithms favor similarities within the top stack frames, which are closest to the crash. An advantage unique to the hash based algorithm, is that each new crash can be put in a group without comparing with other call stacks.

### 10.1.5 Consequences of a two-level grouping approach

For performance reasons it might be necessary to do an initial grouping before starting call stack grouping [33]. This is identified as being more important in an automatic crash reporting system than when grouping crashes from fuzzing. If the amount of crashes from fuzzing reaches a level of millions per day, a two-level approach would probably be necessary also in this context.

There are several options for how to perform a first level of grouping. A crash-type can contain all crashes from one module or one function. If all crashes are from the same version of the program, function offsets can also be used. Otherwise the offsets may vary between versions. A hash algorithm or heuristics depending on a few stack frames could also be used to decide the crash-type.

When the first level of grouping is based on the top stack frame, crashes can only be grouped if their crash locations are equal. This might hide relations between crashes from different crash-types. It is argued that grouping crashes caused by different bugs has a more negative impact on bug-fixing time than separating crashes belonging to the same bug [33]. On the other hand, if an existing relation is not known by analysts, it could require deep analysis of both groups before it is realized that they are caused by the same bug [50]. In this perspective, it might be cost-effective to show these relations without affecting the existing groups.

One solution could be to allow crashes to belong to more than one

group. This could be confusing, and it is argued that crash groups should be kept static throughout the lifetime of a bug [33]. Instead relations could be identified without influencing the grouping. Topological relations in the crash graph could be identified manually or automatically. Levenshtein distance can be used as a measure of similarity across groups. The distance between two groups can be calculated as the distance between the representative traces of each group. As shown in Section 10.2 these relations can be used for prioritizing crashes for bug fixing.

### 10.1.6 Suggested changes to the distance algorithm

There may be no ideal algorithm for grouping crashes [39]. A more general result is therefore how a particular grouping can be evaluated. Calculating Levenshtein distances between crashes is shown to be a useful measure of similarity. Trace diversity and Silhouette values can be calculated using any similarity measure, so these methods are valid even if the distance algorithm is modified or replaced.

A distance algorithm can be used to affect grouping, but also to identify similarities between groups. A distance algorithm works by comparing individual elements of a sequence. When used for call stack comparison, the elements are stack frames. The stack frames compared in this thesis contain a module name, a function name and a function offset. Different LD algorithms were implemented by including or omitting the offset when comparing stack frames. In both cases the distance between individual stack frames were either one or zero.

A compromise could be to use a distance of 0.5 between stack frames with equal function names and different offsets. This would reflect that these frames are neither equal nor completely different. Similarly, stack frames from different functions within the same module or source file could be given a distance of e.g. 0.8. This would add more granularity to the distance. Table 10.2 illustrates this approach.

Modules	Functions	Offsets	Distance
unequal	-	-	1.0
equal	unequal	-	0.8
equal	equal	unequal	0.5
equal	equal	equal	0.0

Figure 10.2: A granular distance between individual stack frames

The values of the distance constants could be fine tuned to fit the properties of different programs. They could even be customized for individual functions or modules. For example different offsets in a large function could count more than different offsets in a small function. This would reflect the fact that different parts of a large function could be completely unrelated. Such a modified distance algorithm could give a more true measurement of similarity between call stacks.

The distance algorithm used in this thesis gives an equal weight of all similarities within the top ten frames. Only the top frame is given

extra weight, when used to decide the crash-type in a two-level approach. Differences in the call stack far from the crash are unlikely to say anything about the reason for the crash [78]. This is accounted for to some extent by comparing only the top ten stack frames.

However, when there is a hard boundary at the tenth frame, it might have unwanted side-effects. A difference at the eleventh frame is not counted, but a difference at the tenth frame counts as much as a difference in the second frame. A solution could be to use a linear weighting of stack frame distances based on proximity to the crash. The following example of crash graph analysis gives a supporting argument for this approach.

Empirical studies could explore the possible effects of the proposed changes to the distance algorithm. The LD algorithm can be modified to use different distance values between individual elements. This is e.g. necessary when comparing with representative traces [33]. However, no exact implementation is suggested for favoring similarities near the top stack frame. This is a topic for future research.

### 10.1.7 Applications of crash graphs

The crash graph in Figure 9.3 showed an aggregated view of all unique call stacks. Figure 10.3 shows six related EIP groups represented by black nodes. The two EIP groups to the left have a crash location in the function `gs_type1_interpret`. The crash graph shows that this function is also in the second or third stack frame of four additional EIP groups.

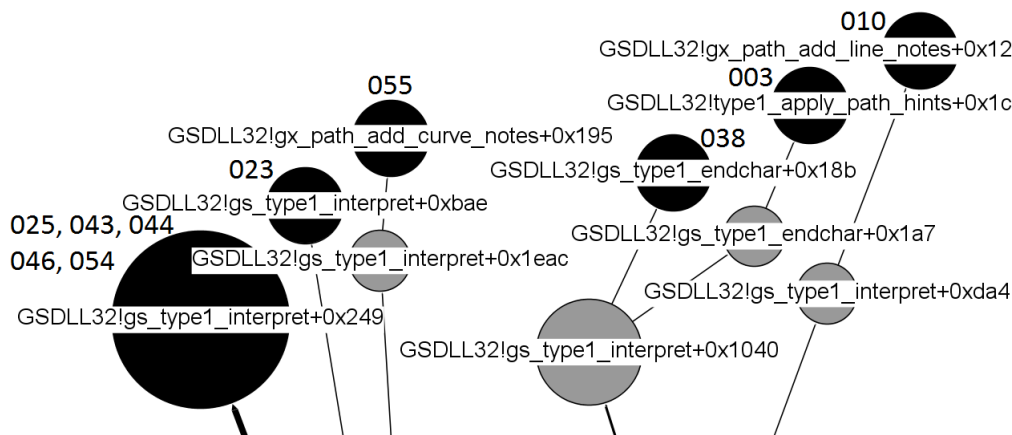


Figure 10.3: Related crashes shown in a crash graph

The nodes in Figure 10.3 represent strictly unique stack frames by comparing both function names and offsets. Nodes could also be created by comparing only the function name. Alternatively a function node could be a supernode containing subnodes of offsets. A granular distance value between stack frames could also be used to visualize their similarities. Proximity of nodes in the visualization could be influenced by the granular distance between node pairs.

Table 10.1 shows how algorithm 10 grouped the crashes from Fi-

Figure 10.3 by splitting one EIP group and joining others. 003 and 038 are grouped, reflecting that these crashes are topologically close in the crash graph. 023 and 055 are grouped with their respective subsets of the left-most crash location. This means that five different call stacks sharing crash location are divided into two groups. One that is more similar to 023 and one that is more similar to 055. The resulting four groups contain only crashes involving `gs_type1_interpret`. No other call stacks were similar enough to join these groups.

Group	Crashes	TD
1	003, 038	2
2	010	0
3	023, 043, 044	0
4	025, 046, 054, 055	1

Table 10.1: A selection of the groups created by algorithm 10

The crash graph in Figure 10.4 shows the complete call stacks of the crashes from Figure 10.3. Inspection of the crash graph shows that differences in the eighth, ninth and tenth stack frames are affecting the grouping. If these differences are not counted, it would result in only two groups. One for the crashes in the upper right and one for the crashes in the lower right. This supports the use of a weighted distance algorithm, making differences near the top stack frames count more than differences further down the call stack.

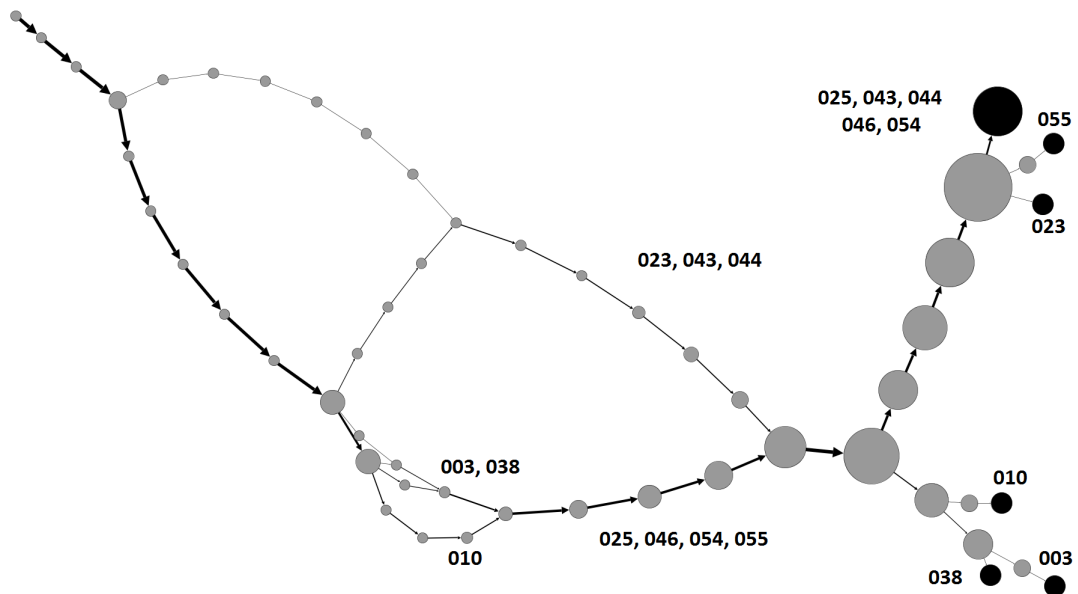


Figure 10.4: Crash graph of call stacks related by `gs_type1_interpret`

Regardless of how grouping is performed, this example shows how a crash graph can be useful to understand relations between different call stacks. A distance algorithm may provide a quantitative measure of

similarity, but a crash graph provides more qualitative information about similarities and differences.

Distance algorithms are suitable for measuring similarities between sequences, and call stacks are ordered sequences of stack frames. A crash graph on the other hand, is a network of stack frames created from multiple call stacks. Instead of using a distance algorithm, there might be graph algorithms suitable for finding relations and grouping crashes based on the complete graph of all call stacks.

A stack frame in this context is a link between two program points. The first point is where a function call was performed. More accurately, the return address points to the instruction directly following the call instruction. The second point is a location inside the called function. The program points are historic in the sense that the crash dump shows that these function calls have been executed.

Using dynamic analysis, more historic program points can be logged. An extension of the call stack is a complete call graph of all function calls and returns. An execution trace might further extend this by logging basic blocks or individual instructions.

There is no qualitative difference between these program points and the stack frames from call stacks. They all give a chronologically ordered sequence of executed code, i.e. control flow. A crash graph might therefore be extended to contain all function calls or even a complete execution trace.

The question is whether this will help developers understand and fix bugs. If the data can be collected automatically and be presented in a manageable way to an analyst, it is possible. This could be answered by more research on crash graphs.

### 10.1.8 Crash dump analysis of related crashes

Section 8.2 described how input from related crashes in Ghostscript 6.51 produced related crashes in version 9.04. The new crashes were 100% reliable, i.e. they were verified in ten out of ten times. This indicates that the stability of the program has increased over ten years.

However, the crashes in the new program were classified as more severe than the original crashes. The crashes in the old program were all named `PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull` while the new crashes were `PROBABLY_EXPLOITABLE` and `EXPLOITABLE`. The classifications indicate that these particular input files produce more security critical situations in the new program than the old program. This serves as a reminder that exploitability is not a property of corrupted data, but how the program handles the corrupted data.

The descriptions given by `!exploitable` for the new crashes are quite explanatory. The crashes occur in the context of an indirect call. Figure 10.5 shows the code from the crash dumps. The first crash happens by the dereference of an invalid function pointer, `EAX` (line 1). Taint analysis performed by `!exploitable` within the basic block shows that `EAX` may control code flow if it contains a valid address.

For the second crash, `EAX` does contain a valid address. The data at

```
1 mov edx,dword ptr [eax] ; crash 1: TaintedDataControlsCodeFlow
2 push ecx
3 push esi
4 call edx ; crash 2: StackCodeExecution
```

Figure 10.5: Automatic analysis of related crash locations

this location is assigned to EDX and used as a function pointer. If EDX was assigned an invalid address, it would generate an exception on the call instruction (line 4). However, EDX contains a stack address. This leads to stack memory being interpreted as code. Then an unhandled exception occurs while running code on the stack.

The two related crash locations are at offsets close to each other within the same function. However, comparing the interval between offsets is not an accurate measure of similarity between two program locations. Two program locations in a function are closely related if they are topologically close in the graph of basic blocks. For example if they belong to the same or adjacent basic blocks. This metric could influence the granular distance between two stack frames.

This example shows that automatic analysis of crash dumps may provide useful information for understanding how crashes occur. It also shows relationships between classifications and descriptions. The fact that a read access violation might be changed into an exploitable situation is discussed in Section 10.2.

### 10.1.9 Summary of answers to RQ1

This section has showed that bucketing of crashes can be performed in multiple levels of an expanding sort tree. Condensing rules can be applied by grouping crashes based on crash location and call stack.

The crash location can be used as an initial level of grouping to determine the crash-type. Call stack comparison can divide a crash-type into subgroups of similar call stacks. If possible in practice, a one-level approach could provide more flexibility to a grouping algorithm, allowing it to group crashes with different crash locations.

Two possible changes to the distance algorithm were proposed. The first was to use a granular distance between individual stack frames. The second was to give a linear weighting of distances based on proximity to the top stack frame. It was also suggested that a distance algorithm can be used to identify relations between different groups of crashes.

Crash graphs were used to illustrate relations between stack frames and call stacks. It was also suggested that more control flow could be added by including a complete call graph or even basic blocks or individual instructions.



## 10.2 RQ2: How should crashes be prioritized?

In the context of field crash reports and fuzzing results, the number of crashes can be large. A given amount of time is required to locate and fix the corresponding bugs. Prioritizing crashes can help developers fix the most important bugs first.

Prioritizing program crashes is a matter of identifying certain properties and finding an algorithm for how crashes should be ordered based on those properties. For example one crash might be more security critical than others. Another crash might be more frequent and affect more users. A third property to consider is whether a crash is related to already fixed bugs.

### 10.2.1 Security implications of crashes

If a program is to be used in a context where input might come from untrusted sources, a severity estimate of crashes can be an important priority factor. If some crashes stand out as more likely to be exploited, these should be fixed first. A software company might also consider an *out-of-band* [84] security update for exploitable bugs.

Table 8.1 showed that a large portion of the crashes were classified as Unknown by !exploitable. Such a classification indicates that a crash may or may not be exploitable. Section 9.2 showed how taint analysis can provide more certainty to a classification. By showing if user input may influence the value of the operands responsible for the crash, the severity estimate can be raised or lowered. Neither of the methods can give a certain estimate of exploitability.

A disadvantage of using execution traces for analysis of read access violations, is that it cannot answer what would happen if the read operation was valid. A read access violation might lead to an exploitable situation if the invalid address could be changed into a valid address. The first crash shown in Figure 10.5 is a read access violation classified as Probably Exploitable. The second crash showed that a more severe situation occurred when EAX contained a stack address.

This relationship was found because different fuzzing inputs generated the two situations. Depending on black-box fuzzing to uncover such situations is inefficient because of its random nature. Instead it could be possible to perform such an analysis automatically for all read access violations. For instance, if a crash is classified as not exploitable by CrashWrangler, it is recommended to run the test case again allowing read access violations by dynamically allocating memory at the invalid address [89].

White-box fuzzing can be used to automatically change the input so that a read operation is valid. If a solution is not found, an alternative is to allocate memory at the invalid address. This can simulate heap spraying [83] which was briefly described in Section 6.3. The method can also apply to whole-system binary instrumentation, allowing one or more read access violations. This can extend an execution trace to contain the

possible control flow and data flow after the crash.

Conservative severity estimates are recommended [14]. For example !exploitable classifies a stack buffer overflow as Exploitable without knowing if the crash can be controlled by user input. It might be possible to know that a given bug is exploitable, e.g. if *Proof of Concept* [63] exploit code has been developed, or the bug has been exploited by malware. However, it might be impossible to prove the opposite, that a given bug cannot be exploited. In any case, developers should not put more effort into determining the security implications of a bug than the effort needed to fix it.

### 10.2.2 A weighted priority model based on a strategic policy

Security implication is only one possible factor for prioritizing crashes. Another factor is how many users may be affected by a given crash. To estimate this, frequency and reliability of crashes are useful metrics. Knowledge about the program and user statistics can also help estimate if a given crash will be common or rare in practical use.

Frequency analysis of field crash reports can give a good indication of the affect on users. However, some users choose not to submit crash reports, or they cannot because their system is not connected to the Internet. The frequency of a crash in fuzzing results might also estimate the possible affect on users. Reliability of crashes can be seen in combination with frequency. An unreliable crash is less likely to occur in normal use than a reliable crash. Reliability can also affect severity estimates, because reliable crashes are more likely to be exploited [32].

Different parts of a program serve different purposes. A bug may be considered more important if it occurs in a vital part of the program. It can for instance be in a function called by many other functions. The function can be necessary for the program to run, or it can be part of a rare functionality used by only a few customers. A software company could maintain a strategic priority policy, guiding developers toward the most important crashes.

An expanding sort tree as shown in Figure 10.1 can be used to prioritize different program parts. A frequent source file or function could be prioritized, such as `gstype42.c` or `append.simple`. A priority policy could for example give a low priority to all crashes in a module that is rarely used. Available resources can also affect the priority policy. For instance if the developer of a complex module is temporarily unavailable, it might be cost-effective to postpone analysis of a crash in this module until the developer is available. The discussed factors indicate that there is no ideal algorithm to prioritize crashes. A priority algorithm should rather depend on a strategic priority policy.

Table 9.2 shows how different factors could affect crash priority. For example nine crash locations are in one of three `append`-functions in the source file `gstype42.c`. Two of these are classified as Exploitable. The crash locations are sorted first by frequency, then by severity and last by reliability. Instead a weighted priority model could be used. This could

give a *priority score* to each crash based on differently weighted metrics and heuristics. The weighting should be determined by strategic priorities.

A priority score can be calculated for groups of crashes by e.g. using the maximum or mean score of all crashes in a group. This could produce an ordering of the 30 call stack groups discussed in Section 10.1 based on strategic priorities. The score could also be calculated for each node in an expanding sort tree. The priority score of a node could be based on the score of the child nodes. It could also be influenced by a priority policy for each level in the tree, e.g. by giving different weight to different modules.

### 10.2.3 Prioritizing crashes related to fixed bugs

Frequency analysis can be used e.g. to prioritize a module, source file or function for bug fixing. Even though there might be multiple bugs causing the crashes, there might be synergy effects when focusing on one part of the program. Understanding the algorithms and data structures involved is necessary to implement a reliable fix in source code.

Table 9.2 shows that 29 unique call stacks have a crash location in the most frequent source file. This is nearly half of all unique call stacks. One could assume that fixing these crashes would require nearly half of the total fixing time. Automatic grouping caused these crashes to be put into six groups containing all the 29 call stacks and no other crashes. The six groups are only 20% of the 30 groups created by algorithms 6 and 9 (Appendix E).

The six groups are created because there are six significantly different call stacks. This might correspond to six different bugs. There can also be fewer bugs triggered in six different ways. It is likely that analysis of the crashes will show a pattern that will make fixing time decrease from group to group. Prioritizing the most frequent crashing functions could therefore be an effective way of quickly removing the most common problems.

The root cause analysis in Section 9.3 targeted this specific source file. Crashes 052 and 045 were shown to be caused by the same bug. If these are grouped, there are only five groups from the same source file. There are seven additional call stacks in the same group as 045. They should also be considered when locating and fixing this bug. After that it would be natural to prioritize the four remaining groups before the 24 unrelated groups. The recently acquired program knowledge could then be efficiently reused. This would probably reduce the analysis time of the related groups.

As discussed in Section 10.1, related groups can be identified automatically by using a distance algorithm on call stacks. A crash graph can also be used for this purpose. A threshold can be used when comparing unfixed groups and fixed groups. Only groups that are more similar than the threshold are treated as related. The calculated similarity can be used as a weighted factor in a priority algorithm.

### 10.2.4 Summary of answers to RQ2

This section suggests that strategic priorities should be reflected in a weighted priority model. Possible metrics can be frequency, reliability,

severity estimate and identified relations to fixed bugs.

Methods for determining exploitability were discussed. Automatic classification can give an indication. Taint analysis can provide more certainty. Proof-of-Concept exploit code or observed exploitation by malware can prove exploitability. However, proof of the opposite might be infeasible, and exploitability should never be ruled out as being impossible. Hence, conservative estimates are recommended.

Frequency analysis of crash locations can help identifying program parts that should be prioritized for analysis. This should be balanced with strategic priorities, e.g. the estimated user impact of crashes in specific program parts. A reliability measurement can help estimate the user impact of individual crashes, both concerning the probability that a crash will occur during ordinary use and the possible security implication of the crash.

### 10.3 RQ3: How should crashes be fixed?

Section 9.3 demonstrated different methods of root cause analysis. The methods were crash dump analysis, dynamic analysis using execution traces and input analysis. The different methods provide different information and have different advantages and limitations.

The underlying cause of memory corruption bugs should be analyzed in order to fix them completely without introducing new bugs [68]. To understand how program crashes caused by access violations should be fixed, it is necessary to understand their possible causes at machine-level.

#### 10.3.1 Causes of access violations

Access violations are caused by illegal operations on virtual memory addresses. The failing instruction may attempt to read, write or execute a memory address without having the required permissions. In most cases the address generating an exception is non-existing, i.e. it is not a valid memory address for any operation.

Before a memory address is used in an illegal operation, it originates from a set of valid instructions. The address may be composed from a number of valid sources and arithmetic operations. Using an execution trace with logged operands, it is possible to track memory assignments backward in time starting from the failing instruction. The sources of the illegal operand are stored in valid memory locations. A general approach of root cause analysis is therefore to ask why these locations contain values that result in illegal memory access.

For example, an invalid address  $X$  might be read via an address pointer  $Y$  pointing to the memory location containing  $X$ . If it is possible to find an assignment to this memory location before the crash, it might explain why  $X$  is invalid. On the other hand, if this memory location is not previously written to by the program, it can mean several things. The data could be uninitialized. It could be assigned by a different process via shared

memory. Also the address pointer  $Y$  could be corrupted, and it must be checked for previous references to the memory location containing  $Y$ .

To summarize, access violations can have different causes. The following is a non-exhaustive list:

- Source(s) corrupted by legal, unintended write operation(s):
  - explicit overwrite, e.g. caused by incorrect use of data pointers
  - implicit overwrite, e.g. buffer overflow caused by improper boundary checking or size calculations
  - changing an array index to point outside the lower or upper boundaries of the array might cause the dereference of an invalid address
- Source(s) not corrupted, but used in an unintended manner
  - use of uninitialized data
  - null pointer dereference
  - reuse of a data pointer after its data has been freed
  - violation of access rights, e.g. trying to write to memory that is not writable

### 10.3.2 Crash dump analysis

A crash dump can reveal the location of a crash, the type of unhandled exception and a stack trace. Section 3.3 showed how function variables and arguments of stack frames can be inspected. If heap memory is dumped, values of dynamically allocated data can be analyzed. As shown in Section 9.3, even stack frames of returned functions can be identified by traversing frame pointers. These stack frames can also be inspected for local variables and arguments. This suggests that the complete stack memory should be included by default in a crash dump.

The main limitation of a crash dump is that it shows only the last program state. For example, the last values of variables and arguments can be inspected, but previous values are not present. Also the stack frames of returned functions might have been overwritten by subsequent function calls. The crash dump analysis of crash 052 performed in Section 9.3 identified a function stack frame destroyed by a buffer overflow. However, the stack frame of the function responsible for the overflow was not present in the crash dump.

The main advantage of a crash dump is that it can be generated automatically when a crash occurs. It can also be analyzed automatically to some extent. For example a call stack can be generated by using WinDbg. It could also be possible to display available stack frames of returned functions automatically. In the case of a corrupted call stack, Section 9.3 demonstrated recovery of unaffected stack frames.

This process is more challenging to automate, because the chain of frame pointers may be broken. Another aspect is that some stack frames

do not contain frame pointers. This can make it impossible to know if a given return address is part of the current call stack or if it belongs to a returned function. In any case, each identified return address is a historic program point which may give a clue about the path leading to a crash.

To summarize, crash dumps can be used efficiently to determine where and how a crash occurred, but they cannot in general tell why a crash occurred. Available stack frames give only small pieces of control flow and data flow. The crash location and stack trace might guide developers in the right direction [78], but for deeper understanding of a crash, dynamic analysis may be required [68].

### 10.3.3 Dynamic analysis

Section 9.3 demonstrated how an execution trace with logged operands can be analyzed by searching for important addresses. First the address of a stack location containing a return address was searched for to find an unintended overwrite of this stack location. Then key variables were inspected by searching for code addresses of instructions operating on the respective variables.

This method can be used generically to find code that has an impact on a program crash. For example, in the case of a null pointer dereference, the method can be used to find code that assigned null to the pointer. It can also find previous use of the pointer, e.g. if it has been passed as a function argument or if it has been checked for a null value. Because a null pointer dereference occurred, one or two code elements are missing from the particular program execution. The pointer should either have been checked for a null value before it was used, or some part of the program should have assigned a valid address to it. These two program points cannot be found using this method, because they were not executed.

Differential debugging can on the other hand identify code that should have been executed. For example the good trace might initialize data which the bad trace for some reason fails to initialize [68]. This could be identified as a diverging point. Comparing operand values at this part of the trace might uncover a logical error in the program. This could be e.g. improper validation of input data or improper handling of an identified error.

The information given by dynamic analysis is complete control flow and data flow. It shows the code paths executed before a crash and the previous values of variables and arguments. It is possible to use a debugger to inspect runtime values of variables and which code paths are taken. However, an input file can cause the crashing function to be called several times before the crash [68]. In addition, there might be variations in control flow and data flow between program runs. If an unreliable crash is analyzed with a debugger, the crash might not occur.

The advantage of an execution trace is that it is possible to track data flow and control flow from the crash and backward in time. While a debugging session must be restarted to analyze previous program states, an execution trace enables repetitive analysis of one concrete program run. A limitation is the time and disk space needed to produce even short

execution traces [21]. Tracing can be started in a function near the top of the call stack. TEMU can also start tracing when reaching an address for the Nth time. This will generate a smaller trace, but information important to the crash might also be lost. *Slicing* and *aligning* traces can be used to isolate code important for a crash [68]. This can enable faster variable inspection, but there is also a risk of missing code operating on a memory location.

Execution traces have an additional limitation compared to crash dumps and dynamic analysis using a debugger. Even though operand values are logged, the complete memory content is not logged for each program point. A debugger can inspect all valid memory locations at any given point, not only the memory involved in the current instruction. A crash dump is a snapshot of important parts of process memory which can be inspected using a debugger.

For instance, the call stack is present in the stack memory region of a crash dump. In an execution trace, the call stack is not stored at any point in the trace. However, by reading the execution trace from beginning to end, the call stack can be reconstructed for all points. Similarly, the values of accessed memory locations can be saved in a *simulated memory layout*. As different parts of memory are being read and modified, the simulated memory layout can provide a more complete picture of the program state at any given point of execution.

### 10.3.4 Input analysis

An advantage of analyzing fuzzing results compared to field crash reports, is that the input generating the crash is available. When doing mutational fuzzing, two inputs are known for each crash, i.e. the fuzzing template and the mutated input.

Section 9.3 showed how comparison of input files might help explaining why a crash occurred. In the example, it was identified that a mutation was performed inside an embedded font description. For a simple data model, it is necessary to compare inputs to know how a mutation affected input. If multiple mutations are performed per iteration, it can be useful to isolate the mutation or mutations responsible for generating the crash. If it is possible to use a complete data model, the fuzzer can tell exactly what kind of data was mutated for each iteration generating a crash.

In the data set of this thesis, equal crashes were produced by different mutations of different fuzzing templates. A subject for future research could be *differential input analysis*. Analyzing different inputs giving the same crash could help developers understand the bug. This subject is possibly related to *differential slicing* [49]. A noteworthy distinction is that plain input analysis does not require execution tracing nor taint analysis. Instead, the mutated inputs generating crashes might give developers clues about what kind of invalid input the program should be able to handle. This can help them expect the unexpected.

### 10.3.5 Consequences of unreliability

One could argue that there are no false positives when analyzing crashes [14]. Regardless of how reliable a crash is, it has evidently happened at least once and should be analyzed. However, it can be challenging to collect more information from the crash if it cannot be reproduced.

When crashes cannot be reliably reproduced, dynamic analysis can be challenging. For instance, it might be necessary to produce several execution traces until the crash is captured in a trace. If such a trace can be created, it might be more valuable than the trace of a 100% reliable crash.

Before producing execution traces, it could be beneficial to analyze the variations between inputs producing the same crash. It might be possible to show that some inputs are more reliable than others. Doing an automatic verification of all inputs in a crash group could point out the most reliable input. This input could be the best candidate for dynamic analysis. Such a process was performed for all crashes in this thesis. It was used to calculate max reliability in Table 9.2. However, all produced execution traces came from crashes that occurred exactly once during fuzzing. Hence, only one input candidate was available from each analyzed group.

The test cases in a crash group could be used to verify that a fix in source code really removes the problem created by the mutated inputs. The identified reliability of inputs should be considered in this verification process. Since testing shows the presence, not the absence of bugs [24], such verification can only be used to detect if there is still a problem. It cannot prove that the bug was fixed properly, nor that the changed source code did not introduce any new bugs. If an input generated a crash only two out of ten times in the original program, it must be run a minimum of five times on the fixed program to serve as an indication that the bug was fixed. Because of the uncertainty of the reliability measurement it should probably be run more.

It is also not sufficient to test only one input from the crash group. One input may produce different crashes, and some inputs could produce new crashes that are only discovered after the first problem is fixed. This suggests that all inputs in a crash group should be tested for the verification of a fix. In an automatic crash reporting system, this can be impractical. Instead verification can be based on the fact that no crash reports from the updated program version are similar to the old crash reports of fixed bugs.

### 10.3.6 Early versus late error handling

Many program paths may lead to the same crash. The crash graph in Figure 10.4 showed a possible example of this. It can be understood as the same bug being triggered from different parts of the program via function calls. One bug might also cause crashes in several locations depending on input. This is shown in the crash graph as two paths resulting in three crash nodes each. It was not proven that these six crashes were caused by only two bugs. However, call stack based grouping made this probable.

Figure 10.6 shows how a bug can have multiple triggers and multiple



symptoms. A symptom in this context is a program crash. There is no fixed relationship between triggers and symptoms.

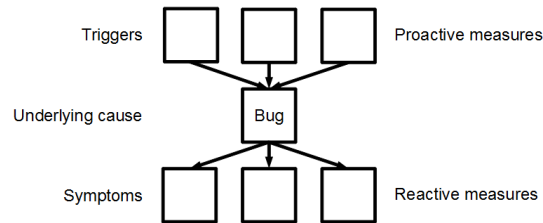


Figure 10.6: Triggers and symptoms of a bug

Depending on the bug and the program, a bug fix could involve both triggers and crash locations in addition to the location of an identified bug. Proper input validation can be seen as *proactive* bug elimination. This could prevent corrupt data of reaching complex data processing code, more liable to contain errors. It does not mean that the actual bug should not be fixed. It should be fixed in order to make the program more robust. A supporting argument is that there might be other code paths that can trigger the bug [68], still not identified by fuzzing nor field crash reports.

Handling corrupt data at an early stage can help prevent unanticipated problems. As a *reactive* measure, the crashing code could also be made more robust, e.g. by validating pointers before using them. Crash dump analysis might give enough information to fix code at a crash location. Dynamic analysis may be needed to locate the underlying bug. Dynamic analysis can be supported by crash graph analysis, differential debugging and input analysis to identify different triggers. In addition, methods not described in this thesis can be used. For example *static analysis* tools can be updated with the new bug pattern to identify other potential triggers and to reveal similar bugs in other parts of the program [18, 45].

Section 9.3 suggested a bug fix for crashes 052 and 045. The fix involved changes to four functions. This was necessary to pass information about a buffer limit from the function allocating the buffer to the function writing to the buffer. It might also be necessary to review other parts of the program using these functions, i.e. other potential triggers. As a reactive measure, a stack cookie [23] could be added by the compiler, and DEP [16] combined with ASLR [53] could mitigate control flow hijacking [70]. These features are security mechanisms that can make it more difficult for an attacker to exploit memory corruption bugs [34]. They would not remove the bug, and the program would still crash. However, the possibility of system compromise could be reduced.

Fixing a bug at the location of the crash can be seen as removing a symptom, not the cause. If the crash happens because of some corrupted data structure, like uninitialized data, then the underlying problem lies in the creation of the data. If it is possible to create corrupted data, then this data could possibly be used by a different function and produce a previously unknown crash.

### 10.3.7 Summary of answers to RQ3

Memory corruption bugs should be completely understood at machine-level in order to find appropriate solutions in source code.

Crash dump analysis and dynamic analysis can provide the data flow and control flow needed to understand the underlying cause of a crash. Using a machine-level debugger or execution traces, memory locations and individual CPU instructions can be inspected. A main advantage of execution traces with logged operands is that they allow inspection of control flow and data flow from the crash location and backward in time. This was described as a generic approach for root cause analysis.

A bug may have several triggers and several symptoms. All triggers should be identified and data corruption should be detected before the data is used. Early input validation and error handling might prevent other unknown bugs from being triggered. Reactive measures can be applied to minimize the consequences of bugs.

Reviewing information about crashes can help developers realize weaknesses in design. Insufficient input validation can be caused by developers not anticipating all possible error situations. Crash reports and fuzzing results might show developers unexpected situations and corner cases, so that the program can be redesigned to be more robust.

## 10.4 Automatic program crash analysis

The analysis of crashes performed in this thesis is a combination of automatic and manual methods. It is in general impossible to automate the whole process from crash to fix. This is supported by the results of related research [68, 43, 32]. However, certain parts of the process can be automated. Identifying these parts and letting automatic and manual analysis work together might be crucial to reducing bug fixing time.

Fuzz testing and crash reporting systems are automatic methods for identifying program bugs. These methods can identify problems that were missed during development and automated static analysis. Program crashes are only symptoms of program bugs. The challenge is therefore how to identify the root causes of crashes, and how to correct source code using that knowledge. If the main goal is to reduce the time needed to fix each bug, a key factor should be to automate as much as possible of the process.

Because of the large number of crashes, automatic grouping might reduce the necessary time of manual analysis. Automatic classification, frequency analysis and reliability analysis can help developers prioritize crashes. Automatic identification of relations between groups might speed up crash analysis and reduce bug fixing time.

Section 9.1 showed how call stack reconstruction from an execution trace can be automated. Section 9.3 showed a potential for automatic recovery of stack frames from a crash dump. Other methods could be made semi-automatic. For example variable inspection in execution traces and comparing mutated input files with their fuzzing templates.

**Part VI**  
**Conclusion**



# Chapter 11

## Conclusion

During the work on this thesis, different methods for analyzing program crashes were tested. The methods were applied and evaluated in the context of reducing bug fixing time.

### 11.1 Major contributions

This thesis shows that different methods of program crash analysis are applicable to analyze fuzzing results. Evaluation methods for grouping algorithms are taken from the context of automatic crash reporting systems and applied to the context of fuzzing. Proposed grouping algorithms from crash reporting systems and fuzzing contexts are compared.

Grouping algorithms using Levenshtein distance as a threshold proved to be competitive compared to the other algorithms tested. A two-layer grouping approach should only be used if it is necessary for performance reasons. To compensate for the bias created by an initial level of grouping, relations between crash groups could be identified automatically by comparing representative traces. These relations could also be used as a prioritization metric. Two suggested modifications to the distance algorithm might give a more accurate measure of similarities between call stacks. The modifications were inspired and supported by empirical studies of crash graphs as described in this thesis.

Prioritization of crashes for bug fixing is discussed with the goal of reducing the possible impact on users. In this context, security implications and probability of occurrence are two important factors. This thesis suggests how a priority model could be based on differently weighted metrics according to a strategic policy.

Crash dump analysis performed in this thesis showed a potential for automatically recovering available stack frames from a corrupted call stack. It was also identified that the stack frames of recently returned functions could be recovered automatically. In the case study, these stack frames were used to recover a corrupted crash location. This suggests that a larger part of the stack memory region should be included in a stack dump by default.

Miller et al. [68] demonstrated how root cause analysis using Bit-Blaze [22, 82] can help to determine exploitability. This thesis demonstrates

how it can be used to suggest a fix in source code. The generic approach of finding the source of memory corruption in an execution trace is simpler than the methods described by Miller et al. The case study performed in this thesis, showed that it can be performed automatically with only minimal knowledge of assembly code. The method can be used to point out in source code where the corruption took place and automatically reconstruct the call stack at the moment of corruption.

## 11.2 Summary of results

Program crashes from a real program were produced. The crashes were produced by using dumb, black-box, mutational fuzzing. The fuzzing statistics showed that most crashes were rare. Still new unique crashes were produced in a regular manner. This confirms the results of C. Miller, presented at CanSecWest in 2010 [67].

Automatic classification and bucketing of crashes was performed by using the !exploitable Crash Analyzer. 87% of the total crashes were classified as Unknown. Under 10% of the unique crashes were classified as Probably Exploitable or Exploitable. There was a large variation of frequency and reliability among the crashes. An inherent instability of the target program added uncertainty to the reliability measurement.

Different call stack based grouping algorithms were tested. The resulting groupings were evaluated by comparing trace diversity and Silhouette values as proposed by Dhaliwal et al. [33] in 2011. The algorithms using a similarity threshold performed best. The threshold was adjusted with the goal of minimizing the amount of groups while producing low trace diversity and high Silhouette values for the groups.

It was discussed how expanding and condensing rules can affect grouping. Bucketing could be performed in a strictly expanding sort tree with multiple levels. Condensing rules could be based on crash location and call stack similarities.

Comparison of grouping algorithms showed that small variations in the algorithms could have a large impact on grouping. For the data set of this thesis, the algorithms performed overall better when comparing function names of stack frames and ignoring the offsets.

It was also identified that a two-level approach can have a negative effect on grouping if the initial grouping criteria are too discriminating. Crashes in a group should be more similar to other crashes within its group than to crashes in other groups. To ensure this, crashes belonging to the same bug might be divided into more than one group. This effect can be compensated for by automatically identifying relations between crash groups.

Two modifications of the distance algorithm were suggested. The first is to calculate a granular distance between individual stack frames. This could reflect that some stack frames are related, although not equal. The second is to favor similarities near the top stack frame. No concrete implementation for these modifications are described. Future research

could answer if this would give a more accurate measurement of similarity between call stacks.

Analysis of Crash Graphs described by Kim et al. [50] was demonstrated as a method for identifying relations between crashes. Graphs were produced for the complete set of call stacks and for selected crash groups. Weighting of nodes and edges was used to identify call stack relations. Frequent stack frames and call paths were identified as large nodes and thick edges. Related crashes were identified as being topologically close.

Frequency and reliability of crashes was compared and discussed as metrics for prioritizing crashes. Taint analysis was demonstrated as a complementary method to automatic severity estimates based on crash dumps. Relations between fixed and unfixed crash groups was proposed as an additional prioritization metric.

A *weighted priority model* based on different metrics was suggested. A *priority score* could be calculated for crash groups to help prioritize the groups for bug fixing. The score could be used to create a prioritized list of crash groups. It could also be calculated for each node in an expanding sort tree for strategic prioritization of different program parts.

Call stack reconstruction was performed by using execution traces. Recovery of available stack frames from a crash dump was also demonstrated. These methods give information about control flow leading to crashes. Root cause analysis was demonstrated by using a crash dump, an execution trace and input analysis. The analysis resulted in a suggested solution in source code.

A generic method of root cause analysis using execution traces was demonstrated. By searching for a specific memory location it was possible to identify code that corrupted memory at this location. Data flow was analyzed by inspecting instruction operands in the trace. Differential debugging was discussed as a method for identifying diverging points of execution. This could e.g. show where a variable should have been initialized.

Advantages and limitations of crash dumps and execution traces were discussed. A crash dump is fast to produce, but gives less information about control flow and data flow. An execution trace provides more information, but requires more time and disk space to produce. Input analysis was discussed as a method for understanding the trigger of a bug. Because of unreliability of crashes, it might be insufficient to analyze only one of the inputs causing a particular crash. In addition, different inputs causing the same crash represent different triggers. *Differential input analysis* was suggested as a method for comparing different triggers of the same bug.

Two improvements were suggested for the use of dynamic analysis using execution traces. The first is to use dynamic instrumentation to allow one or more read access violations to pass, simulating heap spraying. This could answer what would happen if an invalid read address can be controlled by user input. The second is to reconstruct a *simulated memory layout* of the virtual address space of a process by reading the values of operands. This is similar to reconstructing the call stack at a given point in

the trace.

When fixing a bug, it was suggested that triggers and crash locations should be considered in addition to the actual bug. Crash dump analysis can help analyze and fix code at a crash location. Dynamic analysis can help understand the underlying cause of the crash. In particular, this thesis shows how the link between memory corruption and crash can be found by using an execution trace with logged operand values. Crash graphs, differential debugging and input analysis are methods that can help understanding the underlying cause and different ways to trigger a bug. Lastly, it was argued that proactive measures are better than reactive measures when it comes to developing reliable and robust software.

This thesis has showed a potential for automization in many aspects of program crash analysis. First automatic crash reporting and fuzzing were described as automatic methods for collecting program crashes. Semi-automatic code coverage analysis and mutational fuzzing was demonstrated. In addition, potential for automization was identified for all three research questions.

### 11.3 Critical evaluation

In this thesis some methods were described and discussed but not demonstrated with examples.

#### 11.3.1 Evaluation of methods

Differential slicing [49] was introduced in 2011, but the article was not discovered until after the empirical work of this thesis was nearly finalized. Differential debugging could have been applied to the case study of root cause analysis. However, this approach was described in detail in case studies by Miller et al. [68] so enough empirical results were available to support the discussion of how this method fits in the context of fixing bugs.

The method of searching for addresses in an execution trace was not compared to slicing and aligning traces to identify influences of a crash. The methods could have been compared regarding correctness and time consumption. Only local variables were inspected using address search. The demonstrated method depends on instructions operating on the variables. Global variables and heap data could possibly be inspected in the same way, but this was not attempted. Tracing and taint analysis of all 62 crashes was not performed. While it could have been feasible, it was not assessed to be relevant for the research questions.

An effort could have been made to link discovered bugs to historic bug fixes through analysis of the open source code repository. This could evaluate the performance of grouping algorithms by knowing exactly which crashes belonged to which bugs. It could also give answers about bug fixing time and how the bugs were fixed. However, the open source development did not include an automatic crash reporting system. Hence, the historic bug fixes were not based on crash dumps, but rather source level debugging and source code auditing.



Crashes were the only bug symptoms used in this thesis. Error messages and program behavior during fuzzing was ignored. As described early in this thesis, memory corruption may have other symptoms such as program freeze and resource exhaustion. These symptoms and bug classes were not considered.

When comparing call stack based grouping algorithms, the computational complexity of the algorithms was not considered. This could be an important factor in a practical implementation. The two-level approach suggested by Dhaliwal et al.[33] created competitive groupings, and this was created for performance. However, the computational cost of using a one-level approach was not assessed. The hash algorithm of !exploitable was described as efficient because it does not require comparison with other call stacks, but the algorithm complexity was not assessed.

It was decided to calculate Silhouette values by comparing to all other crash groups. If Silhouette validation was performed within a crash-type, the threshold could possibly be increased. This could create less groups while keeping Silhouette values high. If there were relations across crash-types, these could be identified by calculating Levenshtein distances between representative traces of the groups.

### 11.3.2 Validity of results

The results of this thesis are based on a concrete set of crashes. The crashes were produced and analyzed by using techniques similar to previous studies [67, 33, 50, 68]. All methods are described, so the results may be replicated. The set of fuzzing templates and crashes were produced in a random manner, so the exact same data set cannot be reproduced.

The reliability measurement may be inaccurate. However, it served to show that measuring crash reliability can be difficult. Also, it described in principle how reliability can be used in combination with frequency and severity to prioritize crashes.

The data set did not include the relations between crashes and bugs. Neither did it contain information about bug fixing time. This may be a threat to *internal validity*. To compensate for this, the discussion was supported by results of related research [33, 68]. However, results and conclusions about different methods are presented without knowing for certain if these methods would reduce bug fixing time for this data set.

This may also be a threat to *external validity* because it is possible that these results and conclusions cannot be generalized to other data sets. The results, conclusions and modifications to current methods given in this thesis should therefore be taken as suggestions, not exact proof of how the research questions should be answered.

## 11.4 Future work

This thesis does not address analysis of system crashes. Nor does it analyze crashes in programs running managed code. A subject for further research could be to apply the discussed methods to these kinds of crashes. The

methods might also be applicable to other architectures, such as ARM [80] and x86-64 [5]. Future research could also evaluate the suggested changes to existing methods.

A concrete implementation of a modified distance algorithm should be developed. It should favor similarities near the top stack frame and use a granular distance between individual stack frames. Different criteria for deciding the crash-type in a two-level grouping approach could be compared. The initial level of grouping should reduce the need for call stack comparison while not being too discriminating. The new algorithms should be tested on different data sets, including crash reporting systems where information about bug fixing time is known. On such a data set, it could also be evaluated if relations to fixed bugs could be an efficient metric for prioritizing crash groups.

Crash graphs could be expanded with more control flow data from execution traces. Relations between nodes in a crash graph could be visualized by calculating granular distances between stack frames or by using supernodes and subnodes. It might also be possible to explore graph algorithms that take the complete crash graph of all call stacks as input. The application of such algorithms could be both grouping crashes and identifying relations between crashes or groups of crashes.

An analysis not performed in this thesis is taint propagation of all crashes. The results could be compared to classification made by !exploitable. Such a complete analysis was also not performed by Miller et al. [68] This could evaluate the relevancy and effectiveness of automatic classification based purely on crash dumps.

Whole-system instrumentation or other methods could be used to allow one or more read access violations and observe if this could result in a write access violation. This could provide more certainty to severity estimates of this particular kind of crashes. Reconstructing the memory layout in an execution trace may allow more detailed inspection of data flow.

Differential input analysis could be explored as a method for identifying different triggers of a bug. This could be a complementing method to differential slicing.

## 11.5 Final remarks

It can seem like a paradox that programs still crash after decades of development and improvements in the field of computer science. This thesis aims to enlighten the subject and answer how developers can investigate and fix these issues. However, program crash analysis can only help remove the symptoms of inherent weaknesses of computer architectures. Future operating systems, programming languages and hardware platforms may be more robust from design. The ultimate goal must be to avoid the possibility of program crashes, making this field of research obsolete.

# Appendices



## Appendix A : Mutation of a fuzzing template

Figure A.1 shows a random mutation inside a PostScript file used as a fuzzing template. The characters marked in bold font are the mutated bytes. The mutation replaced the original characters with non-printable NULL characters (ASCII value 0x00).

---

```
... (line 8333)
%beginsfnt
truedictknown type42known or( %endsfnt)exch fcheckload
/FontMatrix [1 0 0 1 0 0] def
/FontBBox[2048 -319 1 index div -441 2 index div 2147 3 index div \
 1985 5 -1 roll div]cvx def
/FontType type42known{42}{3}ifelse def
systemdict/product 2 copy known{get dup(LaserWriter II)eq \
exch(LaserWriter IIG)eq or version(2010.113)eq and not}{pop pop \
true}ifelse{/UniqueID 16#00D2761B

def}if/sfnts[<

000100000009000900090009

6376742043851A570000009C00000648
... (line 9217)

0811182CAFA4EF614C8FFB7E165D60000000000000000000000000000000072FFDDDD1C...
```

---

Figure A.1: Mutation of a Type42 font description in a PostScript file



## Appendix B : !exploitable rules

CLASSIFICATION	DESCRIPTION	FINAL
NO_EXCEPTION	The current event is not an exception	true
EXPLOITABLE	Exception from code running in the Stack	true
EXPLOITABLE	Illegal Instruction Violation	true
EXPLOITABLE	Privileged Instruction Violation	true
EXPLOITABLE	Guard Page Violation	true
EXPLOITABLE	Stack Buffer Overrun (/GS Exception)	true
EXPLOITABLE	Heap Corruption	true
EXPLOITABLE	Kernel Mode Data Execution Protection Violation	true
EXPLOITABLE	Data Execution Protection Violation	true
PROBABLY_EXP	Data Execution Protection Violation near NULL	true
EXPLOITABLE	User Mode Write AV	true
PROBABLY_EXP	User Mode Write AV near NULL	true
EXPLOITABLE	Write AV in Kernel Memory	true
EXPLOITABLE	Write AV in Kernel Mode	true
EXPLOITABLE	Kernel Mode Read AV at the Instruction Pointer	true
EXPLOITABLE	Read AV at the Instruction Pointer	true
PROBABLY_EXP	Read AV Near Null at the Instruction Pointer	true
EXPLOITABLE	Kernel Read AV on Control Flow	true
EXPLOITABLE	Read AV on Control Flow	true
PROBABLY_EXP	Read AV on Control Flow near NULL	true
PROBABLY_EXP	Read AV on Block Data Move	true
PROBABLY_EXP	Kernel Memory Read AV on Block Data Move	true
PROBABLY_EXP	Memory Read AV on Block Data Move	true
PROBABLY_EXP	Tainted data controls Code Flow	true
PROBABLY_EXP	Tainted data controls subsequent Write Address	true
PROB_NOT_EXP	Read AV near NULL	false
PROB_NOT_EXP	First Chance Kernel Read AV in User Memory	false
PROB_NOT_EXP	First Chance Kernel Write AV in User Memory	false
PROB_NOT_EXP	Integer Divide By Zero	false
PROB_NOT_EXP	Float Divide By Zero	false
UNKNOWN	Breakpoint	false
UNKNOWN	BugCheck	false
UNKNOWN	Possible Stack Corruption	false
UNKNOWN	Kernel Read Access Violation near NULL	false
UNKNOWN	Tainted data is used in a subsequent Block Data Move	false
UNKNOWN	Memory Read Access Violation on Block Data Move	false
UNKNOWN	Tainted data is used as arguments in a Function Call	false
UNKNOWN	Tainted data may be used as a return value	false
UNKNOWN	Tainted data controls Branch Selection	false
UNKNOWN	Read Access Violation	true
UNKNOWN	Write Access Violation	true
UNKNOWN	Data Execution Protection Violation	true

Table B.1: !exploitable rules derived from source code





## Appendix C : Derivation of an expression for $Z(i)$

The following sentences derive an expression (C.3) for the accumulative amount of unique crashes, as described in Section 4.3. Crashes are represented by numbered balls.

$Z(i)$  is the amount of unique numbers picked after  $i$  iterations. It is defined recursively by:

$$\begin{aligned} Z(0) &= 0 \\ Z(i) &= Z(i-1) + u(i) \quad \forall i \in \mathbf{N} \setminus \{0\} \end{aligned}$$

where  $u(i)$  is the probability that the  $i^{\text{th}}$  ball has a new unique number greater than zero, i.e. it belongs to an empty bucket.

The probability of picking a ball with the number  $j$  is constant throughout the iterations. It is given by:

$$p_j = \frac{q_j}{Q}$$

If bucket  $j$  is empty,  $j$  must not have been picked in any of the previous iterations. That gives the following expression for the probability that bucket  $j$  is filled with its first ball on iteration  $i$ :

$$p_j \times (1 - p_j)^{i-1}$$

The expression evaluates to  $p_j$  when  $i = 1$  and decreases for each iteration. This reflects that all buckets are empty in the beginning. Also, a higher  $q_j$  will result in a faster decrease of the probability that bucket  $j$  is still empty.

$u(i)$  is the sum of probabilities for all buckets.  $u(0)$  is undefined and  $u(i)$  is given by:

$$u(i) = p_1 \times (1 - p_1)^{i-1} + \dots + p_k \times (1 - p_k)^{i-1}$$

The expression for  $Z(i)$  then becomes:

$$Z(i) = Z(i-1) + \sum_{j=1}^k (p_j \times (1 - p_j)^{i-1})$$

which can be written as a sum of sums:

$$Z(i) = \sum_{n=1}^i \left( \sum_{j=1}^k (p_j \times (1 - p_j)^{n-1}) \right) \quad \forall i \in \mathbf{N} \setminus \{0\}$$

The expression can be simplified by extracting the sums for each bucket.

$$Z(i) = Z_1(i) + \dots + Z_k(i)$$

$$Z(i) = \sum_{n=1}^i \left( p_1 \times (1 - p_1)^{n-1} \right) + \dots + \sum_{n=1}^i \left( p_k \times (1 - p_k)^{n-1} \right)$$

$$Z_j(i) = p_j + p_j \times (1 - p_j) + \dots + p_j \times (1 - p_j)^{i-1} \quad (\text{C.1})$$

$$(1 - p_j) \times Z_j(i) = p_j \times (1 - p_j) + \dots + p_j \times (1 - p_j)^i \quad (\text{C.2})$$

Subtracting equation C.2 from C.1 results in the following:

$$\begin{aligned} p_j \times Z_j(i) &= p_j - p_j \times (1 - p_j)^i \\ Z_j(i) &= 1 - (1 - p_j)^i \end{aligned}$$

The expression for  $Z(i)$  is the sum of  $Z_j(i)$  for all  $k$  buckets:

$$\begin{aligned} Z(i) &= \sum_{j=1}^k Z_j(i) = \sum_{j=1}^k \left( 1 - (1 - p_j)^i \right) \\ Z(i) &= \sum_{j=1}^k 1 - \sum_{j=1}^k (1 - p_j)^i \\ Z(i) &= k - \sum_{j=1}^k \left( 1 - \frac{q_j}{Q} \right)^i \quad \forall i \in \mathbf{N} \end{aligned} \quad (\text{C.3})$$

Expression C.3 is defined for all positive integers and zero.  $Z(0)$  evaluates to zero because the sum evaluates to  $k$  if  $i = 0$ . When  $i$  approaches infinity, the sum converges to zero, and  $Z(i)$  converges to  $k$ :

$$0 < p_j < 1 \quad \forall \text{ integers } j \in [1, k] \Rightarrow \lim_{i \rightarrow \infty} Z_j(i) = 1 \quad \wedge \quad \lim_{i \rightarrow \infty} Z(i) = k$$

## Appendix D : List of unique crashes

Table D.1 lists all unique crashes produced in Ghostscript 6.51. The major and minor hashes were recalculated based on correct symbols.

Table D.1: Unique crashes in chronological order

No	Name
001	UNKNOWN_TaintedDataPassedToFunction_0x6a591068_0x571a5a16
002	UNKNOWN_TaintedDataControlsBranchSelection_0x355b2a31_0x42182e6c
003	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x171d7605_0x052e6a21
004	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x3b341d03_0x13633d7f
005	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x13554811_0x4f4a1368
006	UNKNOWN_TaintedDataReturnedFromFunction_0x475f0d70_0x40414f11
007	UNKNOWN_TaintedDataReturnedFromFunction_0x475f0d70_0x40414f10
008	UNKNOWN_ReadAV_0x32542264_0x5c193e2a
009	PROBABLY_EXPLOITABLE_ReadAVonControlFlow_0x19654837_0x5464783c
010	UNKNOWN_TaintedDataControlsBranchSelection_0x5d500e11_0x441a5d65
011	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x7c744f06_0x280d7f63
012	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x77437800_0x607b1e7e
013	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x4a707b3d
014	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x4e342d79
015	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x7f67273e
016	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x2c6e3503
017	UNKNOWN_TaintedDataControlsBranchSelection_0x4e68534b_0x370a5e62
018	UNKNOWN_TaintedDataControlsBranchSelection_0x355b2a31_0x196b4045
019	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x314c1206
020	UNKNOWN_ReadAV_0x3e064a33_0x314c4606
021	UNKNOWN_ReadAV_0x13554811_0x4f4a1368
022	UNKNOWN_TaintedDataControlsBranchSelection_0x66637c08_0x58793b67
023	UNKNOWN_TaintedDataPassedToFunction_0x22163c41_0x3c6e0e64
024	PROBABLY_EXPLOITABLE_ReadAVonControlFlow_0x19654837_0x61166701
025	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x22163c41_0x02514a2a
026	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x7c744f06_0x1d7f605e
027	UNKNOWN_TaintedDataControlsBranchSelection_0x4e68534b_0x7d02657a
028	UNKNOWN_TaintedDataReturnedFromFunction_0x25072f39_0x4f164e5b
029	UNKNOWN_ReadAV_0x424d592f_0x60044050
030	UNKNOWN_TaintedDataPassedToFunction_0x75662e1d_0x6c437725
031	UNKNOWN_TaintedDataControlsBranchSelection_0x59496a7f_0x4147175f
Continued on next page	

APPENDIX D. LIST OF UNIQUE CRASHES

Table D.1 – continued from previous page

No	Name
032	UNKNOWN_TaintedDataPassedToFunction_0x7d250953_0x0715452e
033	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x71714c0e
034	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x44035333
035	UNKNOWN_ReadAV_0x3e064a33_0x314c2906
036	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x2c653567
037	UNKNOWN_ReadAV_0x3e064a33_0x2c6e0903
038	UNKNOWN_TaintedDataControlsBranchSelection_0x4c483f7d_0x32003858
039	UNKNOWN_ReadAV_0x4e68534b_0x370a3b62
040	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x2c633567
041	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x1e6e3503
042	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x034c1206
043	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x22163c41_0x3c370e64
044	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x22163c41_0x78334a32
045	EXPLOITABLE_WriteAV_0x52757738_0x2c655267
046	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x22163c41_0x074c680d
047	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x7c744f06_0x28057f63
048	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x4a7b7b59
049	UNKNOWN_ReadAV_0x3e064a33_0x4e344879
050	EXPLOITABLE_WriteAV_0x515a442d_0x67287c10
051	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x1c6a275a
052	EXPLOITABLE_ReadAVonIP_0x5b525c40_0x4b051e6a
053	UNKNOWN_TaintedDataControlsBranchSelection_0x3e064a33_0x39521519
054	PROBABLY_NOT_EXPLOITABLE_ReadAVNearNull_0x22163c41_0x2a44760a
055	PROBABLY_EXP_TaintedDataControlsWriteAddress_0x5562253a_0x3f6d4276
056	UNKNOWN_ReadAV_0x3e064a33_0x4e347979
057	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x4e3f2d1d
058	UNKNOWN_ReadAV_0x23381138_0x536b3b49
059	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x7f6c275a
060	UNKNOWN_TaintedDataControlsBranchSelection_0x52757738_0x4e392d1d
061	UNKNOWN_ReadAV_0x3e064a33_0x4a701e3d
062	UNKNOWN_ReadAV_0x424d592f_0x60047250

## Appendix E : Call stack grouping

Table E.1 shows the 30 groups produced by algorithms 6 and 9 from Table 9.1. These are LD algorithms using a threshold of two.

Crash function	Unique crashes	TD
append_simple (1)	013-016, 019, 020, 033-035, 037, 041, 042, 049, 053, 056, 061	0.5
append_simple (2)	036, 040, 045, 048, 051, 057, 059, 060	0
gs_type1_interpret (1)	023, 043, 044	0
gs_type1_interpret (2)	025, 046, 054	0
pdf_write_embedded_font (1)	009, 024	0
s_DCTD_process	005, 021	0
igc_reloc_struct_ptr (1)	006, 007	0
gs_text_replaced_width (1)	029, 062	0
pdf_font_notify_proc (1)	011, 047	0
append_simple (3)	017, 039	0
append_simple (4)	027	0
igc_reloc_struct_ptr (2)	028	0
gs_text_replaced_width (2)	008	0
pdf_font_notify_proc (2)	026	0
s_filter_close (1)	002	0
s_filter_close (2)	018	0
pdf_put_colored_pattern (1)	004	0
pdf_put_colored_pattern (2)	012	0
append_component	058	0
append_outline	052	0
cos_dict_elements_write	032	0
gs_type1_endchar	038	0
gx_path_add_line_notes	010	0
gx_path_add_curve_notes	055	0
pixel_resize	030	0
ref_param_make_int	001	0
refset_null_new	050	0
restore_finalize	022	0
s_zlib_free	031	0
type1_apply_path_hints	003	0

Table E.1: Call stack grouping of unique crashes



# Bibliography

- [1] DynamoRIO Dynamic Instrumentation Tool Platform.  
<http://dynamorio.org> (Accessed April 2012).
- [2] GDB - The GNU Project Debugger.  
<http://www.gnu.org/software/gdb> (Accessed April 2012).
- [3] Gephi, an open source graph visualization and manipulation software. <http://gephi.org> (Accessed April 2012).
- [4] Hex-Rays IDA Pro.  
<http://www.hex-rays.com/idapro> (Accessed April 2012).
- [5] Intel® 64 and IA-32 Architectures Software Developer Manuals.  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html> (Accessed April 2012).
- [6] QEMU open source processor emulator.  
<http://wiki.qemu.org> (Accessed April 2012).
- [7] zynamics BinDiff. © Copyright 2004 to 2011 by Google Inc.  
<http://www.zynamics.com/bindiff.html> (Accessed April 2012).
- [8] zynamics BinNavi. © Copyright 2005 to 2011 by Google Inc.  
<http://www.zynamics.com/binnavi.html> (Accessed April 2012).
- [9] The Type 42 Font Format Specification. Adobe Developer Support - Technical Note # 5012, July 1998. [http://partners.adobe.com/public/developer/en/font/5012.Type42\\_Spec.pdf](http://partners.adobe.com/public/developer/en/font/5012.Type42_Spec.pdf) (accessed April 2012).
- [10] GNU Ghostscript 6.51, © Artifex Software, July 2001.  
<http://sourceforge.net/projects/ghostscript/files/gnu-gs/6.51/> (Accessed April 2012).
- [11] Portable Document Format, PDF, (ISO 32000-1), 2008.
- [12] Ghostscript Home Page, June 2011.  
<http://pages.cs.wisc.edu/~ghost/> (Accessed April 2012).
- [13] The C++ Resources Network, 2011.  
<http://www.cplusplus.com> (Accessed April 2012).

## BIBLIOGRAPHY

---

- [14] S. Lambert G. Wroblewski A. Abouchaev, D. Hasse. Crash Course - Analyze Crashes to Find Security Vulnerabilities in Your Apps. MSDN Magazine, November 2007. <http://msdn.microsoft.com/en-us/magazine/cc163311.aspx> (Accessed April 2012).
- [15] Aleph1. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996. <http://phrack.com/issues.html?issue=49&id=14> (Accessed April 2012).
- [16] Starr Andersen and Vincent Abella. Memory Protection Technologies. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3, September 2004. <http://go.microsoft.com/fwlink/LinkId=28022> (Accessed April 2012).
- [17] Chris Anley, Jack Koziol, Felix Linder, and Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, Inc., New York, NY, USA, 2007.
- [18] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, September 2008.
- [19] Bernhard Beckert and Claude Marché, editors. *FoVeOOS'10: Proceedings of the 2010 international conference on Formal verification of object-oriented software*, Berlin, Heidelberg, 2011. Springer-Verlag.
- [20] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.
- [21] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments, VEE '06*, pages 154–163, New York, NY, USA, 2006. ACM.
- [22] BitBlaze: Binary Analysis for Computer Security. <http://bitblaze.cs.berkeley.edu/> (Accessed April 2012).
- [23] Brandon Bray. Compiler Security Checks In Depth. MSDN, February 2002. <http://msdn.microsoft.com/en-us/library/Aa290051> (Accessed April 2012).
- [24] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.
- [25] Bruno Cabral and Paulo Marques. Exception handling: A field study in java and .net. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 151–175. Springer Berlin / Heidelberg, 2007.



- 
- [26] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 1066–1071. ACM, 2011.
- [27] Apple Computer. TrueType Reference Manual, December 2002. <http://developer.apple.com/fonts/TTRefMan/> (Accessed April 2012).
- [28] Jean-Pierre Courtiat, Piotr Dembinski, Gerard J. Holzmann, Luigi Logrippo, Harry Rudin, and Pamela Zave. Formal methods after 15 years: status and trends: a paper based on contributions of the panelists at the formal technique '95 conference, montreal, october 1995. *Comput. Netw. ISDN Syst.*, 28:1845–1855, October 1996.
- [29] Microsoft Security Engineering Center (MSEC) Dave Weinstein, Jason Shirk. The History of the !exploitable Crash Analyzer. Microsoft TechNet Blogs, Security Research & Defense, April 2009. <http://blogs.technet.com/b/srd/archive/2009/04/08/the-history-of-the-exploitable-crash-analyzer.aspx> (Accessed April 2012).
- [30] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [31] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Proceedings of the 19th international conference on Computer aided verification, CAV'07*, pages 20–36, Berlin, Heidelberg, 2007. Springer-Verlag.
- [32] Jared D. DeMott, Richard J. Enbody, and William F. Punch. Towards an automatic exploit pipeline. In *Proceedings of the 6th International Conference for Internet Technology and Secured Transactions (ICITST-2011)*, pages 323–329. IEEE, Dec 2011.
- [33] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *ICSM*, pages 333–342. IEEE, 2011.
- [34] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.
- [35] Michael Eddington. Peach code coverage tool minset.py. <http://peachfuzzer.com/Tools> (Accessed April 2012).
- [36] Michael Eddington. Peach Fuzzing Platform. <http://peachfuzzer.com> (Accessed April 2012).

## BIBLIOGRAPHY

---

- [37] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4, WSS'00*, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.
- [38] Susan Gerhart, Dan Craigen, and Ted Ralston. Experience with formal methods in critical systems. *IEEE Softw.*, 11:21–28, January 1994.
- [39] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09*, pages 103–116, New York, NY, USA, 2009. ACM.
- [40] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.
- [41] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.
- [42] Danny Goodman, Michael Morrison, and Brendan Eich. *JavaScript® Bible, Sixth Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2007.
- [43] Sean Heelan. Vulnerability detection systems: Think cyborg, not robot. *IEEE Security and Privacy*, 9:74–77, May 2011.
- [44] Mario Hewardt and Daniel Pravat. *Advanced windows debugging*. Addison-Wesley Professional, first edition, 2007.
- [45] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.
- [46] R. Hyde. *The Art of Assembly Language*. Number v. 1 in No Starch Press Series. No Starch Press, 2003.
- [47] Adobe Systems Inc. *PostScript language reference (3rd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [48] Intel Corporation, Santa Clara, California. *Intel 80386 Programmer's Reference Manual*, 1986.
- [49] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11*, pages 347–362, Washington, DC, USA, 2011. IEEE Computer Society.

- [50] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage (practical experience report). In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2011.
- [51] Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9:0934–5043, 1997.
- [52] Joseph B. Kruskal. An Overview of Sequence Comparison: Time Warps, String Edits, and Macromolecules. *SIAM Review*, 25(2):201–237, 1983.
- [53] Lixin Li, James E. Just, and R. Sekar. Address-space randomization for windows systems. In *Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC '06*, pages 329–338, Washington, DC, USA, 2006. IEEE Computer Society.
- [54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.  
<http://www.pintool.org> (Accessed April 2012).
- [55] Microsoft. About PREfast for Drivers. MSDN. <http://msdn.microsoft.com/en-us/windows/hardware/gg487345> (Accessed April 2012).
- [56] Microsoft. About Static Driver Verifier (SDV). MSDN.  
<http://msdn.microsoft.com/en-us/windows/hardware/gg487498.aspx> (Accessed April 2012).
- [57] Microsoft. Blue Screen Data. MSDN. [http://msdn.microsoft.com/en-us/library/ff538869\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ff538869(v=vs.85).aspx) (Accessed April 2012).
- [58] Microsoft. Debugger Project Settings: Program Database Files (C++). MSDN. <http://msdn.microsoft.com/en-us/library/yd4f8bd1.aspx> (Accessed April 2012).
- [59] Microsoft. !exploitable Crash Analyzer. MSEC Debugger Extensions.  
<http://msecdbg.codeplex.com> (Accessed April 2012).
- [60] Microsoft. .Net Framework. <http://www.microsoft.com/net> (Accessed April 2012).
- [61] Microsoft. Windows Debugger (WinDbg). Debugging Tools for Windows. <http://www.windbg.org> (Accessed April 2012).
- [62] Microsoft. Windows Driver Kit (WDK). MSDN.  
<http://msdn.microsoft.com/en-us/windows/hardware/gg487428> (Accessed April 2012).

## BIBLIOGRAPHY

---

- [63] Microsoft. Proof of Concept Code Published Affecting the Remote Access Connection Manager Service. Security TechCenter, Microsoft Security Advisory (921923), June 2006. <http://technet.microsoft.com/en-us/security/advisory/921923> (Accessed April 2012).
- [64] Microsoft. Win32k Insufficient Data Validation Vulnerability, CVE-2009-2513. Security Bulletin MS09-065, November 2009. <http://technet.microsoft.com/en-us/security/bulletin/MS09-065> (Accessed April 2012).
- [65] Microsoft. Simplified Implementation of the Microsoft SDL, November 2010. <http://www.microsoft.com/security/sdl/discover> (Accessed April 2012).
- [66] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33:32–44, December 1990.
- [67] Charlie Miller. Babysitting an army of monkeys - an analysis of fuzzing 4 products with 5 lines of python. Vancouver, March 2010. CanSecWest. [http://securityevaluators.com/files/slides/cmiller-CSW\\_2010.ppt](http://securityevaluators.com/files/slides/cmiller-CSW_2010.ppt) (Accessed October 2011) <http://www.scribd.com/doc/60008912/cmiller-CSW-2010> (Accessed April 2012).
- [68] Charlie Miller, Juan Caballero, Noah M. Johnson, Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis using BitBlaze. Las Vegas, NV, USA, July 2010. Black Hat USA. <http://securityevaluators.com/files/papers/CrashAnalysis.pdf> (Accessed April 2012).
- [69] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07*, pages 89–100, New York, NY, USA, 2007. ACM. <http://valgrind.org> (Accessed April 2012).
- [70] James Newsome. *Detecting and preventing control-flow hijacking attacks in commodity software*. PhD thesis, Pittsburgh, PA, USA, 2008. AAI3365680.
- [71] John Neystadt. Automated penetration testing with white-box fuzzing. MSDN, February 2008. <http://msdn.microsoft.com/en-us/library/cc162782.aspx> (Accessed April 2012).
- [72] Kristen Nygaard and Ole-Johan Dahl. History of programming languages I. chapter The development of the SIMULA languages, pages 439–480. ACM, New York, NY, USA, 1981.
- [73] Oracle. Java™. <http://www.oracle.com/us/technologies/java/overview/> (Accessed April 2012).

- 
- [74] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.
- [75] Sebastian Porst. BinNavi 3.0 Preview: Improved Differential Debugging. Zynamics Blog, January 2010. <http://blog.zynamics.com/2010/01/19/binnavi-3-0-preview-improved-differential-debugging/> (Accessed April 2012).
- [76] Peter Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20(1):53–65, November 1987.
- [77] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2nd edition, 2010.
- [78] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In Jim Whitehead and Thomas Zimmermann, editors, *MSR*, pages 118–121. IEEE, 2010.
- [79] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10*, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.
- [80] D. Seal. *ARM architecture reference manual*. Addison-Wesley, 2000.
- [81] Tom Shanley. *Protected Mode Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [82] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.
- [83] Alexander Sotirov. Heap Feng Shui in JavaScript. Amsterdam, 2007. Black Hat Europe. <http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf> (Accessed April 2012).
- [84] George Stathakopoulos. Security Advisory 979352 – Going out of Band. Microsoft TechNet Blogs, MSRC, January 2010. <http://blogs.technet.com/b/msrc/archive/2010/01/19/security-advisory-979352-going-out-of-band.aspx> (Accessed April 2012).
- [85] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

## BIBLIOGRAPHY

---

- [86] Laura Thomson. Socorro: Mozilla's Crash Reporting System, May 2010. <http://blog.mozilla.com/webdev/2010/05/19/socorro-mozilla-crash-reports/> (Accessed April 2012).
- [87] VMware. Virtualization Basics - Virtual Machine. <http://www.vmware.com/virtualization/virtual-machine.html> (Accessed April 2012).
- [88] Dmitry Vostokov. *Memory Dump Analysis Anthology, Volume 2*. Opentask, 2008.
- [89] Drew Yao. Announcing crashwrangler. SecurityFocus - Focus on Apple, July 2009. <http://www.securityfocus.com/archive/142/504791/30/0/threaded> (Accessed April 2012).