Australian Government

Department of Defence

Defence Science and Technology Organisation

Fuzzing: The State of the Art

Richard McNally, Ken Yiu, Duncan Grove and Damien Gerhardy

Command, Control, Communications and Intelligence Division

Defence Science and Technology Organisation

DSTO–TN–1043

ABSTRACT

Fuzzing is an approach to software testing where the system being tested is bombarded with test cases generated by another program. The system is then monitored for any flaws exposed by the processing of this input. While the fundamental principles of fuzzing have not changed since the term was first coined, the complexity of the mechanisms used to drive the fuzzing process have undergone significant evolutionary advances. This paper is a survey of the history of fuzzing, which attempts to identify significant features of fuzzers and recent advances in their development, in order to discern the current state of the art in fuzzing technologies, and to extrapolate them into the future.

Published by

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: (08) 7389 5555
Facsimile: (08) 7389 6567

AR No. 015-148
February, 2012

Fuzzing: The State of the Art

Executive Summary

Fuzzing is an approach to software testing where the system being tested is bombarded with test cases generated by another program. The system is then monitored for any flaws exposed by the processing of this input. Whilst such a simplistic approach may sound naive, history has shown fuzzing to be surprisingly effective at uncovering flaws in a wide range of software systems.

This combination of simplicity and effectiveness led to the wide adoption of fuzzing based approaches within the software attacker community around the turn of this century. Early fuzzing implementations tended to be relatively simple, constructing test cases from a sequence of random numbers. Driven by a desire to test increasingly sophisticated systems, the complexity of fuzzing implementations has increased to the point where there are now several recognisable classes of fuzzer, each with their own strengths and weaknesses.

With its effectiveness established, it was only a matter of time before fuzzing was incorporated into software development best practices and used as part of a software test and defensive coding regime. Several companies have developed and released commercial fuzzing tool suites, including fuzzing support for large numbers of computer protocols. This, in addition to research activies within both the academic and commercial spheres, suggests that fuzzing techniques will continue to evolve, and fuzzing will remain an important tool for vulnerability discovery in the future.

This paper is a survey of the history of fuzzing, which attempts to identify significant features of fuzzers and recent advances in their development.

THIS PAGE IS INTENTIONALLY BLANK

Authors

Richard McNally
Command, Control, Communications and Intelligence Division

Richard McNally was awarded a B.Sc. Hons in Computer Science in 1999 from James Cook University in Townsville, Queensland. He joined DSTO's Advanced Computer Capabilities Group in 2000 and currently works in the Command, Control, Communications and Intelligence Division's Information Operations Branch.

Ken Yiu
Command, Control, Communications and Intelligence Division

Ken Yiu was awarded a B.Sc (Applied Maths and Computer Science) in 1993, and a B.E. (Elec.)(Hons) from the University of Adelaide in 1994. He joined DSTO in 1994 and has worked in Trusted Computer Systems and Advanced Computer Capabilities Groups. He now works in Information Operations Branch in Command, Control, Communications and Intelligence Division.

Duncan Grove
Command, Control, Communications and Intelligence Division

Dr Duncan Grove graduated with first class honours in Computer Systems Engineering from the University of Adelaide in 1997, and received a PhD in Computer Science from the same institution in 2003. Following his doctoral studies he joined DSTO's Advanced Computer Capabilities Group where he led the Annex long range research task, which developed Multi Level Security devices and next generation networking technologies. He recently joined DSTO's newly formed Parallax Project where he is continuing to apply his interests in computer security.

Damien Gerhardy
Command, Control, Communications and Intelligence Division

Damien Gerhardy was awarded a Bachelor of Computer Science in 1999 from Adelaide University, South Australia and now works in the Information Operations Branch.

Contents

Glossary

Attack Surface That portion of a software system which is exposed to the user. Users can attempt to influence the system's state by varying their interaction with the Attack Surface.

Black-Box This approach considers that the system being tested is a black-box - i.e. that it has visible inputs and outputs and its presence can be determined, but the interior state and implementation of the system are not visible.

Classical Fuzzer A fuzzer which obeys the Black-Box assumption.

EFS Evolutionary Fuzzing System - extends General Purpose Fuzzer (GPF) by using a Genetic Algorithm for data generation.

GPF General Purpose Fuzzer - an early extensible fuzzing framework used as the basis for EFS. GFS can take network packets and generate semi-valid packets based on various methods.

Grey-Box Unlike black-box, some of the internal workings of the system can be ascertained and are used by the fuzzer. However, the information available is incomplete.

Flaw A mistake in the design or implementation of a system that leads to incorrect behaviour. It is commonly accepted that nearly all complex systems contain flaws, though in many cases the incorrect behaviour is only exhibited under some circumstances.

Fuzzer A piece of software used to test systems by presenting the system being tested with inputs produced as the result of of an algorithmic process (including random input).

Highly Structured Input Many systems use multiple passes to process input. For example a compiler will successfully complete the lexical analysis stage before moving on to parsing and then semantic analysis.

Modern Fuzzer A Fuzzer which relaxes the Black-Box assumption and has some method of monitoring the internal state of the system being fuzzed.

Oracle An Oracle in the fuzzing context is a software component that supplies the correct/expected output of an action. It is most often used to validate the output from the System Under Test. See Test Case.

RE Reverse Engineering - The process of deriving the internal structure, processes or data formats of a system or program. Some of the Reverse Engineering tools the fuzzing community builds upon includes debuggers, disassemblers, static/structural analysis tools and code graph visualisation tools.

Semi-Valid Data Fuzzing data which has been post-processed to make it resemble valid data. Examples of this includes data which correctly obey message formats, have correct length fields, or correct checksums.

SUT System Under Test - This is the program or system being tested.

Test Case A collection of input fed to the System Under Test in order to test it. A test case can succeed or fail, with failure indicating the presence of a flaw. Good test cases (when manually crafted) include the expected output value of the test in the case (Myers 2004); this is optional for fuzzers.

White-Box The fuzzer has access to and uses a relatively complete understanding of the internal operations of the SUT in the fuzzing process.

Vulnerability This is a flaw which is exploitable. Software containing a vulnerability can be made not only to crash, but to execute commands under the control of an attacker.

1 Introduction

Fuzzing is an approach to software testing whereby the system being tested is bombarded with test cases generated by another program. The program is then monitored, in the hope of finding errors that arise as a result of processing this input.

In the early days of fuzzing these test cases consisted of a simple stream of random bytes. Although an approach based on directing random data towards a program's interfaces might seem naive, it proved to be surprisingly effective at uncovering bugs (Miller, Fredriksen & So 1990). It is perhaps even more surprising that fuzzing techniques continue to be effective to this day, as attested to by the level of fuzzing related activity on security mailing lists like BugTraq[1].

This paper discusses the history of fuzzing, from its roots in academia, through its enthusiastic adoption by the black-hat community, to its acceptance as part of corporate software testing best practices. It begins by looking at the evolution of fuzzing in section two. Section three presents a fuzzer anatomy and guide to terminology commonly used in the field. Section four presents a discussion of relative merits of fuzzing and section five discusses the different types of fuzzer. Section six describes the authors' experiences in attempting to use some of the existing fuzzers and frameworks. Section seven summarizes the state of the art and attempts to extrapolate future trends in fuzzing.

While this work attempts to combine input from a variety of sources, the input is predominantly drawn from academic papers and tools and documentation from the black-hat community. Summaries of many of these papers and tools are presented in Appendices A and B respectively. A small number of examples using these tools are also documented in Appendix C. Given the vast array of fuzzing tools available (and the limited nature of some of the proof-of-concept fuzzers which only illustrate a single concept), only the significant fuzzers are covered in this discussion.

[1]see http://www.securityfocus.com/archive/1/description

2 The Evolution of Fuzzing

This section attempts to provide an overview of the history of fuzzing. Rather than attempting to be comprehensive, it is intended to give the flavour of major milestones, advances and trends. The use of randomised test input as a means to stress test software dates back at least as far as 1970 (Hanford 1970), and has also been applied to the problem of hardware validation (Wood, Gibson & Katz 1990). However, this discussion focuses on topics related to the application of fuzzing to software and is restricted to the period from 1990 to early 2011. During this period fuzzing rose from relative obscurity, was widely adopted as an approach to vulnerability discovery by the black hat community, and has become a quality assurance tool commonly used by in-house software testing teams.

2.1 Fuzzing at University of Wisconsin-Madison

Fuzzer development began at the University of Wisconsin-Maddison (UW-Maddison) with the coining of the term “fuzzer” as part of a class project in Professor Barton Miller’s 1988 CS736 class (Miller 1988). Over a span of 18 years, from 1988 to 2006, Professor Miller and his associates performed a series of studies, developing a variety of fuzzers and using them to audit the reliability of applications on various UNIX operating systems. The results of these studies were summarised in a series of four papers. The first two of these papers represent significant milestones in the development of fuzzing, and, as such, are discussed in some detail below. The final two papers, which dealt mainly with porting the methods developed in the first two papers to Windows and MacOS X, are of less interest. It is worth noting however that in each case a significant percentage of the applications tested were found to harbour flaws.

2.1.1 Fuzzing UNIX

Professor Miller’s class project called for students to develop “The Fuzz Generator”. This was to be a tool capable of generating random streams of output suitable for testing the robustness of a variety of UNIX utilities, thus emulating the “fuzz” that line noise threw into serial lines[2]. The resulting program, called `fuzz` was then used to audit the reliability of a suite of UNIX utilities on a variety of UNIX platforms, with the results published in (Miller, Fredriksen & So 1990).

`Fuzz` was, by modern standards, extremely simple. The output it produced consisted of a stream of random bytes. This output was then given to the target application for processing either as input to `stdin` via a pipe or as simulated console input. The simulation of console input was supported by a second tool called `ptyjig`. The System-Under-Test (SUT) was then monitored, waiting for the process to terminate after processing each input. The file system was then examined, looking for `core` files. If a `core` file was found, it indicated that an error had occurred, otherwise there had not been an error. The monitor also incorporated a timeout mechanism, used to identify inputs that hung

[2]The project was apparently inspired when a lecturer at UW-Maddison, connected by modem to the university during a thunder storm, discovered that line noise not only interfered with what he was typing but also caused several UNIX utilities to crash.

the system under test. The result of the audit process found flaws in over a quarter of the tested programs (Miller, Fredriksen & So 1990)[3], thus unequivically establishing fuzzing as a viable means of uncovering flaws.

2.1.2 Fuzzing Revisited

Five years after the first paper Miller et al. published a follow-up paper detailing the results of a repeat of the original UNIX utility audit (Miller et al. 1995). Further, instead of merely repeating the tests from their first study the tests were broadened using three new tools, `portjig`, `xwinjig` and `libjig`. The first of these, `portjig`, simply accepted input via `stdin` and sent it to the program via a network pipe. This allowed network services to be tested in a similar fashion to an application reading from `stdin`. The other two tools, which are addressed below, introduced fuzzing approaches that hitherto had not been seen.

The second tool, `xwinjig`, operated as a man-in-the-middle, forwarding messages between an X Windows server and X Windows application. From this position it was able to generate messages for transmission to either the server or the client application. It incorporated four different methods for generating messages. The four types were:

Random Messages: A stream of random bytes, similar to `fuzz`,

Garbled Messages: An existing message stream was modified through random injection, modification and deletion of bytes,

Random Events: New events that satisfied basic structural requirements were constructed and injected into the message stream, and

Legal Events: New mouse and keyboard events that are completely valid are constructed and injected into the message stream.

According to Miller et al., "Each input type is closer to valid input than the previous one (and therefore potentially tests deeper layers of the input processing code)".

The third tool, `libjig`, was designed to allow the testing of how well applications handle conditions where there were insufficient resources to allocate memory successfully. It did this by intercepting calls to the system's memory allocation routines, such as `malloc` and `calloc`, and randomly failing requests for memory regardless of resource levels.

The audit they conducted using these tools was broken into four parts. The first repeated the testing of command line tools using `fuzz` and `ptyjig`, but expanded the number of platforms from six to nine. The failure rate was between 6% and 43% for each of the operating systems, with only two systems achieving a failure rate of less than 15%[4]. The second set of tests focused on the use of `portjig` for testing network services. The audit was conducted on four OSs, testing each of the services listed in the `/etc/services` file. None of these failed. The third group of tests concerned the use of `xwinjig` to uncover

[3]See entry in Appendix A for more detail

[4]Miller et al. also note that approximately 40% of the bugs found and reported in 1990 were still present in exactly the same form five years later.

flaws in the X Windows server and applications. No bugs were found in the server, but several applications were found to crash or hang. Of particular interest is the number of flaws uncovered using each of `xwinjig`'s different modes. No flaws were uncovered using random messages. Approximately 58% of applications crashed or hung when exposed to various combinations of garbled messages, random events and legal events. 26% of these were susceptible to legal events alone. The final run of tests used `libjig` to randomly fail calls to `malloc`. A total of 53 programs confirmed as using `malloc` were tested; 25 (47%) of them failed.

2.2 Wider Adoption

It is difficult to determine precisely when fuzzing first became used in the black-hat community, as the extent and speed of uptake of fuzzing techniques is not well known. The techniques themselves were readily amenable to bespoke development in scripting languages; dedicated tools and frameworks were not required, although they might have made the task easier and promoted code reuse. It is clear that two important steps in the spread were Aitel's 2002 paper and the release of SPIKE, and the availability of PROTOS at the same time. By 2005 it had transitioned from a little known niche technique to one that was being widely discussed and adopted. It was the subject of many discussions in Defcon 14 and 15, and the Blackhat Briefings (Las Vegas) in 2005, and more subsequently. Vulnerability researchers started using fuzzing as a brute force method for discovering flaws, which they would then analyse to determine whether the flaw represented an exploitable vulnerability. Initially the fuzzing tools developed within the black-hat community were relatively simple, designed to test for classes of simple errors such as uncovering string handling errors. They did this by by passing larger and more complex arguments to programs as command line parameters, WiFi parameters or errors in file formats (Sutton & Greene 2005).

It was quickly realised however that such simple approaches suffered from some serious limitations when it came to uncovering less simplistic flaws. This is perhaps best exemplified by considering what would happen if the SUT being tested by a random fuzzer expected a 32-bit checksum as part of its input. If the SUT validated the checksum before doing any further processing and all of the input bytes were random then there was a 1 in 2^{32} chance that the checksum would be valid for the remainder of the bytes. All remaining inputs would be rejected almost immediately, serving only to evaluate the checksum calculation routines continuously.

One solution to this is exemplified by the PROTOS project, a series of studies conducted at the University of Oulu in Finland (Röning et al. 2002). Starting in 1999, PROTOS was involved in a series of projects established with the goal "Security Testing of Protocol Implementations". In his thesis Kaksonen described a methodology, which he called mini-simulation, that uses a simplified description of the protocol's interactions and syntax to automatically generate system input that *nearly* complies with correct protocol usage (Kaksonen 2001). This approach, which is now generally known as grammar based fuzzing, could give the fuzzer an understanding of the protocol sufficient to allow it control over which aspects of the protocol's correctness were violated. This could be used to ensure that any checksums present within the protocol were always valid, or to systematically vary which rules were broken. Both of these approaches boosted the efficacy of the fuzzing process. Similar, though less well documented approaches were being adopted by the blackhat community.

The year 2001 also saw the first public release of a fuzzing framework, SPIKE, with a subsequent presentation given at Blackhat 2002 (Aitel 2002). SPIKE is a framework that supports *block based fuzzing*. Block based fuzzing allows the structure of input to be specified as a series of layered blocks with varying granularity. The key benefit to this was the ability to build new structures on top of existing ones with less effort. For example, when building a fuzzer that targets Web-application user interfaces, it would be possible to leverage preexisting support for HTTP in HTTP protocol fuzzers to generate the HTTP block/envelope.

Both grammar and block based based fuzzing presented their own drawbacks, most significantly the effort required to develop the requisite grammar describing the syntax and interactions of the SUT. Another approach, known as mutative fuzzing, avoids this effort by taking an existing, valid test case and randomly modifying it to generate a new test case. This process is exemplified by `xwinjig` when operating in garbled message and random event mode.

After completing the original PROTOS project in 2003, a new project, called PROTOS Genome (Viide et al. 2008), began exploring methods that could automatically generate grammars suitable for fuzzing. The system they developed could, when given a representative set of inputs, infer information about the input grammar. This inference process was sufficient to derive grammars for several file archival formats which, when used to generate test cases, proved adequate for discovering flaws in numerous anti-virus products.

Another result of the PROTOS project was a set of testing suites suitable for auditing different implementations of various protocols by various vendors. A variety of protocols were supported including the common LDAP, SNMP, SIP, DNS and HTTP network protocols. The PROTOS SNMP testing suite was released in 2002 containing over eighteen thousand test cases suitable for testing Simple Network Management Protocol (SNMP) implementations. In 2001 the research conducted as part of PROTOS was commercialized to form Codenomicon, the first company devoted to the development of commercial fuzzing tools. Other commercial fuzzing systems have since been released including those by MuDynamics and beSTORM.

The next major advance in fuzzing methodology occurred in approximately 2007 and coincided with increased interest in fuzzing from within the software testing community. Existing approaches to fuzzing had largely been constrained to considering the input and output of the SUT. These new fuzzers, inspired by tools developed within the software testing community, began utilising run-time analysis of the SUT's interal operation to guide the fuzzing process. An early example of this is work done by Sparks et al. using the PaiMei framework to support data-flow analysis (Sparks et al. 2007). Tools like KLEE and SAGE utilise source code access to combine fuzzing with other software testing approaches like binary instrumentation and static analysis to further inform the fuzzing process (Cadar, Dunbar & Engler 2008, Godefroid et al. 2008).

While this concludes the history of fuzzing methodologies, it is worth making two further points. Firstly, although fuzzers have been categorized into a relatively small set of classes, a vast array of different tools and frameworks have been developed, and in many cases released publicly. The table in Appendix B represents just a sampling of significant fuzzers. The second is that the importance of fuzzing has increased such that it is considered best practice to incorporate fuzzers into the software testing and quality assurance processes. This is especially true with respect to security testing as demonstrated by the inclusion of fuzzing within Microsoft's testing practices and published software development lifecycle (Godefroid 2010) (Microsoft 2011).

3 Fuzzer Concepts

3.1 Anatomy of a Fuzzer

A fuzzer can normally be broken into three fundamental components: a fuzz generator, a delivery mechanism, and a monitoring system, each of which is discussed below. Each of these components can be implemented using techniques with varying levels of sophistication and can also support different levels of integration. While some systems, like the configuration fuzzer (Dai, Murphy & Kaiser 2010), resist a clean decomposition according to this taxonomy, the breakdown remains useful. Each of the components is discussed below.

Fuzz Generator: The fuzz generator is responsible for generating the inputs that will be used to drive the System Under Test (SUT). The fuzz generator's output is a series of test inputs, which are subsequently fed to the SUT by the delivery mechanism. A given fuzz generator normally creates inputs that are appropriate for testing specific applications or classes of application. Fuzzers can also use a variety of approaches in the fuzz generation process, each with their own advantages and drawbacks, significantly impacting the effectiveness of the fuzzing process. Many of these approaches are discussed in the following sections. Most of the advances in fuzzing have come as a result of improvements made to the fuzz generator.

Delivery Mechanism: The delivery mechanism accepts system inputs from the fuzz generator and presents them to the SUT for consumption. The delivery mechanism is closely tied to the nature of input that the target application processes. For example, a system which accepts input from a file requires a different delivery mechanism to a system which accepts "user" interaction using a mouse.

Monitoring System: The monitoring system observes the SUT as it processes each system input, attempting to detect any errors that arise. Although the monitor is often overlooked, it plays a critical role in the fuzzing process, directly impacting which classes of error the fuzzer is able to find. Though they have started to receive more attention in recent years, monitoring systems have historically only seen limited advances.

3.2 Fuzzer Terminology

Fuzz The word fuzz is used flexibly within the community, potentially leading to confusion. It is possible to fuzz a target, which is to say use fuzz testing on the target. It is possible to throw fuzz at a target, in which case fuzz refers to one or more of the system input sets generated. Finally, fuzz is sometimes used as the name for the process of mutating legitimate input, such that it makes sense to say that a document has been fuzzed. Fortunately the intended meaning is nearly always clear from the context. This paper uses the term as a verb, describing the process of producing the fuzzing data and applying it to the system being tested.

Classical and Modern Fuzzers A “Classical” Fuzzer is one which obeys the Black-Box assumption: i.e. that the system being tested has visible inputs and outputs, and that its presence can be determined, but the interior workings of the system are not visible. Tools and techniques can be applied to the system (e.g. debuggers to pause the system, and protocol modellers to predict the behaviour of the system), but internal system state is not accessed.

A “Modern” Fuzzer relaxes this assumption and has access to the SUT’s internal state and structures. Some of these techniques can be very sophisticated (including branch analysis tools which feed back into the Fuzz Generator).

Coverage A software system consists of a collection of executable code and data. The code can be broken into a set of basic blocks, each with a single point of entry and a single point of exit. The concept of coverage relates to the number of distinct basic blocks through which execution passes while processing one or more input sets. Coverage is an important concept in fuzzing because of the intuition that basic blocks can contain flaws, so any basic block that remains untested potentially harbours a flaw missed by the fuzzing process.

Depth When software is processing input some segments of the code act as “gatekeepers” for other sections of code, controlling which basic blocks are executed. These can reject a test case before it proceeds to the next stage of processing. For example, code that initially receives the data could reject that input if it encounters non ASCII characters while attempting to read a string, or a later step could reject the input because the resulting string is not one of a predefined set of acceptable strings. The idea of depth corresponds to the number of such gatekeepers that must be passed to reach a particular segment of code, with more gatekeepers corresponding to a greater depth. Significant efforts have been made to develop fuzzing techniques suited to discovering bugs at greater and greater depths within the SUT.

Efficiency There are multiple aspects that contribute to the efficiency of the fuzz testing process but in essence, to be considered efficient, a fuzzer must generate test cases that deliver high levels of coverage at the desired depth with minimal computational effort. When comparing two fuzzers intended to reach a desired depth within a system, the more efficient fuzzer generates less wasted test cases. This means that the test data is not discarded by code at shallower depths, and that code exercised by the test case had not previously been exercised by other test casess. A fuzzer which is highly efficient against one SUT may be inefficient against another, as highly efficient fuzzers are usually tailored to a system.

Black-box, White-box, and Grey-box Testing These terms, taken directly from software engineering parlance, refer to the nature of the information about the operation of the SUT that is used by the testing process. They should not be confused with the purpose behind the testing. Historically black-box testing was the province of black-hat vulnerability researchers and white-box testing was used predominantly in white-hat software testing and quality assurance efforts. White-box and black-box testing are also sometimes characterised as functional testing and structural testing respectively (Kaksonen 2001).

At one extreme is black-box testing, where the system's internal operation is completely opaque and the testing process is limited to observing the system's input and output behaviour. An example of this would be a web-application such as a search engine, because given a search query the internal operation of the web-application remains inscrutable except for what is contained in the output. Although in this case knowledge of the TCP/IP and HTTP protocols do offer at least some insight, these are defined externally to the system itself, and in fact one valid approach to fuzzing is to test compliance with such open standards.

White-box testing lies at the other end of the spectrum. In this case the fuzzer can take advantage of information such as the source code and design specifications of the SUT, allowing greater insights into how the system works. Such information can often be used to increase the efficiency of the fuzzing process. Common examples where such fuzzing is possible include when the SUT is either developed in-house or an open source product.

Grey-box testing covers the circumstances that fall between these two extremes. No universally accepted method for delineating between grey-box and either of the extremes exists. This paper will use the term white-box for any testing process that requires access to source code. Grey-box testing will cover any testing that does not have access to source code, but examines the behaviour of the SUT using any means other than input and output inspection, such as static analysis, reverse engineering, or executing inside a debugger.

4 The Case for Fuzzing

4.1 Why Fuzz?

Fuzzing is just one of many methods that can be used to discover flaws within a software system. Numerous other methods exist, such as source code review, static analysis, development of unit tests, model checking and beta testing. Given the abundance of existing methodologies it is worth looking at why fuzzing has been so widely adopted over the last decade, and in which circumstances it is best applied.

It Works: As has already been mentioned, while automated system input generation may initially seem like a counter intuitive approach to discovering flaws, history has shown it to be surprisingly effective at uncovering non-obvious errors that had been missed by other approaches (Miller, Fredriksen & So 1990). (Godefroid 2010) indicates that a third of Windows 7 bugs between 2007 and 2010 were discovered by the SAGE fuzzer, *after* they had been through (and missed by) the normal software quality control process.

Does not require source code: One of the areas that fuzz testing initially gained traction was within the black-hat community where it was used to search for zero-day vulnerabilities in commercial software products. One of the reasons for this uptake is that many approaches to fuzzing do not require access to the source code for the system under test, which is typically not available for commercial software products. A lack of source code obviously rules out analysis techniques available to white-hats like code review, but it can also hinder the use of approaches like static analysis and model checking. The compilation process can remove symbolic information which assist these processes, especially from production code, which is usually “stripped” of this information, and sometimes even obfuscated to prevent analysis of the program.

Fuzzing can be simple: At its simplest fuzzing can merely involve the generation of random data and feeding that data as input to the system under test. Depending on the nature of the system under test, more complex methods can often yield a more efficient fuzzer, but a basic fuzzer in this fashion is typically relatively simple to develop, especially where the SUT is amenable to a mutative fuzzing approach or a suitable fuzzing framework can be leveraged.

Manual tests are expensive: Historically software quality assurance teams typically relied predominantly on suites of manually developed test cases to test software. Traditional software testing is often positive testing (i.e. testing that features work as specified), rather than negative testing (i.e. testing that the system does not do things that it is not supposed to do), and some flaws cannot be practically found using only positive testing. Fuzzing is one method of cost effective negative testing. As such fuzzing is often used to augment a manually crafted test suite, with the fuzzer able to generate a high volume of tests using methods that probe for unsafe assumptions within the system.

Human testers have blind spots: Software engineering practice has shown that test cases are more effective when written by someone other that the original programmer, since a blind spot in implementation is likely to also be replicated in testing. (Myers 2004) places as two of his princples that “A programmer should avoid attempting to test his or her own program”, and “A programming organization should not test its own programs”. For human testers to be effective, they also need to understand the implemented system, including boundary and corner cases. This is expensive, both in time and conceptual effort, and requires two teams to understand the operation of the system. The tests may be flawed by the testers’ preconceptions of the system (although these can be different from the implementers’). By using random sampling, a significant part of the input state space can be covered without requiring much human code development. This input state space includes items which neither the implementer nor the tester regard as significant, and therefore exercises parts of the attack surface that they might not.

Portability: In many cases fuzzers are not inherently tied to a specific SUT, a fuzzer generated for one system can often be applied to another system. For example a protocol fuzzer such as an HTML fuzzer could be used to test several different web browsers, including different versions and multiple vendors.

Fewer false positives: Historically, static analysis approaches to software testing have tended to swamp the operator with an overly long list of possible flaws, many of which are spurious. Spurious flaws arise because the reasoning processes used for abstract execution are incomplete. Fuzzing, by uncovering flaws through actual execution, only reports flaws that can actually be realised, thus reducing the effort wasted by the tester to analyse false leads. However because of this, it also may not identify as many real flaws as static analysers. Historically however, fuzzing has been unable to differentiate between flaws (which may be benign) and vulnerabilities (which can be exploited by an attacker).

4.2 Strengths and Weaknesses of Classical Fuzzing

Fuzzing’s biggest strength is its ability to augment the efforts of a person testing the software. Even the most basic fuzzer significantly increases the number of input sets that can be used in the validation process.

The earliest fuzzers operated using purely random data. While this initially proved to be surprisingly effective at discovering bugs within a system, it was also quickly realized that random searches were often inefficient. For example a web browser will drop an incoming HTTP message if it does not satisfy the formatting restrictions laid out in the HTTP protocol, so a fuzzer that uses purely random data will only generate a system input set that starts with `"GET / HTTP/1.0\n\n"` 1 in 256^{16} or 2^{24} times. Even when taking into account variations allowed by case insensitivity and HTTP version 1.1, this is still only 1 in 2^{16}, meaning that for every test that reaches the target web application, on average 65535 test cases are rejected by the server before reaching the web application. This realisation gave the motivation for the development of increasingly sophisticated systems for generating input sets that met at least some of the requirements demanded by the target application. Two approaches were adopted for overcoming this problem.

The first was the adoption of mutative approaches which allowed the number and nature of structure violations to be controlled. The second was an increasingly complex support within generative systems for knowledge of input grammars. In time, support for input grammars also passed into the mutative systems, where it allowed for mutation without violating the structural requirements.

Another area where classical fuzzers can be inefficient is if the input contains checksum values. These affect both the generative and mutative approaches, and can render them ineffective unless knowledge of the checksum is incorporated into the fuzzing process.

The final weakness of classical fuzzers is the limited concept of what a flaw is. For example a fuzzer typically cannot detect an access violation when an unauthenticated user is given access to resources that should only be available after a successful login. Instead, they are typically limited to detecting when an application has crashed, or in some cases hung. Several possibilities exist for improving the coverage offered by fuzzers, including enhanced support for detecting execution failures, support for broader classes of errors, and enhancement of applications with increased use of asserts to validate various forms of internal state.

5 A Description of Fuzzer Types

5.1 Fuzz Generators

Numerous methods have been devised for driving the fuzz generation process. This section provides a taxonomy (as per (Takanen, Demott & Miller 2008)) which can be used to describe the operation of fuzz generators. The taxonomy classifies fuzzers according to two distinct axes. The first axis is whether new system input sets are generated by mutating existing input sets or creates new ones from internal logic. The second is the "intelligence" that the fuzz generator uses. These two axes are described below.

There are two fundamental approaches available to Fuzz generators for the production of system input; Mutative and Generative.

Mutative: A Mutative fuzzer takes existing data and distorts it, changing some of the data in order to create a new input set. The second mode of operation supported by `xwinjig` is a good example of this.

Mutators typically have much less intelligence associated with them than Generators, but also require much less human effort to understand protocols and interfaces (substituting computational effort, which can be cheaper).

They can often be applied in cases where the SUT accepts highly structured input; a key reason for the use of a mutative approach is that it allows the fuzzer to easily harness any complex internal structures within the original file without needing to recreate them. This often makes it significantly easier to create a mutative fuzzer that operates at greater depths than it would be to develop an equivalent generative fuzzer. One drawback however is that the coverage achieved by the resultant fuzzer tends to be concentrated near code paths that result from processing the original system input. Thus, it is unlikely that flaws related to functionality for which there are no examples will be uncovered. e.g. If TFTP data is not present in a pcap network traffic capture file, then that protocol may not be fuzzed by a mutator using that pcap.

Generative: A Generative fuzzer creates new system input sets from its own resources. The contents and structure of this input can vary from an unstructured stream of random data to highly structured inputs depending on nature of the interface being fuzzed and the desired depth at which the fuzzer is targeted. Any understanding of necessary format and structure for the input must be incorporated by a programmer, which means they typically require more effort than a mutative fuzzer. However, if a complete understanding of the structure is incorporated, a generative fuzzer can achieve high levels of coverage with excellent efficiency. Any incompleteness in the understanding of the input format can adversely effect the level of coverage, potentially missing flaws that could otherwise be found, or simply failing to exercise much of the SUT's attack surface.

In the simple case, as exemplified by the `fuzz` tool, generation can simply create a stream of unstructured random data.

The trend from Mutative towards Generative fuzzers, especially between 2007-2009, meant that protocol specific or model based generators were written. These were potentially more powerful than mutative fuzzers, but fuzzing a network protocol became more complex. For example, to fuzz a simple network process like FTP, a mutative fuzzer would take a network packet capture and make variations based on this valid data. A generative fuzzer needed to understand a number of protocols to make this work, including DNS. Depending on the type and level of fuzzing required, it might also need to understand TCP state and maintain sequence numbers etc. These operations were certainly possible in some of the fuzzing frameworks, however it meant that beginning to fuzz using generative tools could have a significant start up cost. Once the basic protocol libraries/scripts have been written, code reuse made these operations more useful. This corresponds to the the comments at (Takanen 2009*a*).

The two major classes above can be further characterized by describing the techniques used to produce the random data:

Oblivious: An oblivious fuzzer does not account for the type and structure of information that the system under test expects to receive. In the case of an oblivious mutative fuzzer, the fuzz generator selects bytes within the original test case at random and replaces them with random values. An oblivious generative fuzzer may generate a stream of random bytes. The original `fuzz` tool as described by Miller, Fredriksen & So in their first paper is a good example of an oblivious generative fuzzer (Miller, Fredriksen & So 1990).

Template Based: Template based fuzzers have at least some minimal understanding of the structure accepted by the system. This knowledge was stored in a template that listed the location and type of some components of a well formed test case. Both mutative and generative fuzzers can use such templates to constrain their values used on the regions covered by the template to reasonable values that are less likely to result in rejection during preliminary processing steps. When the SUT expects highly structured input this can increase the depth at which it is possible to fuzz efficiently. One example of a template based fuzzer is `xwinjig` which supported both mutative and generative modes of operation (Miller et al. 1995).

Block Based: This refers to a data representation technique. SPIKE (Aitel 2002) was the original fuzzer which used a block based approach. Initial fuzzers (such as template-based fuzzers) represented data as fields and strings/sequences. SPIKE represented all data as nested data blocks of varying types, making it easier to construct functions that operated on the data injected into the SUT (for example length calculations and check-sum generation).

Grammar Based: A fuzzer that is grammar based incorporates a grammar that covers at least part of the input language accepted by the system under test. Grammars are normally found in generative fuzzers, although they could potentially be applied to mutative fuzzing as well. This is best exemplified with two fuzzers that generate source code for testing compilers (Hanford 1970, Yang et al. 2011).

Heuristic based: A fuzzer that incorporates heuristics can try to make decisions that are "smarter" than random. These heuristics typically try to exploit knowledge of common error sources like edge cases for numbers and Unicode/UTF-8/ASCII transformation errors, integer overflow errors, off-by-one errors and signed/unsigned handling errors.

(Takanen, Demott & Miller 2008) classify the sophistication of a fuzzer's data production capabilities in the ascending order as:

- *Static and random template-based fuzzer.* These are limited to simple request-response protocols and do not incorporate dynamic-functionality.
- *Block based fuzzers* include some rudimentary dynamic content such as check-sums and length values to overcome the inefficiency of generating these randomly.
- *Dynamic Generation or Evolution based fuzzers* have a limited ability to learn a protocol based on feedback from the target system, and can vary the input fed to the SUT based on the data being received from it. These can have an understanding of the program *state*.
- *Model or Simulation based fuzzers* can have a full implementation of a protocol. A model based fuzzer understands the content of the protocol and the sequences of the messages.

All types of fuzzer may use random numbers, however generator based fuzzers are more likely to be (partially) deterministic and use key/boundary values as well. The dependence on random numbers is recognised in Takanen et al. as being simple, however it can be inefficient when exercising control structures (since a test against a 32-bit field may take a long time to find a critical value, especially a check-sum).

5.2 Delivery Mechanisms

Fuzzers typically work by presenting system input to the SUT for it to process. Taking these test cases and presenting them to the SUT as input is the responsibility of the delivery mechanism. Given that the aim is to find flaws in the regular operation of the target program, most delivery systems simulate those normally used by the SUT. As such delivery mechanisms are not normally very complicated, although there are some exceptions. Common delivery mechanisms include:

- Files
- Environment Variables
- Invocation Parameters (e.g. command line and API parameters)
- Network Transmissions
- Operating System Events (includes mouse and keyboard events)
- Operating System Resources

File based fuzzers have an advantage in that all of the relevant fuzzed data is effectively encapsulated in a single entity. Early network fuzzers such as `ProxyFuzz` were quite simple, configured as a network proxy between the SUT and a data source, modifying packets as they flowed through the proxy. This is still more complicated than file fuzzing, because in protocol fuzzing, a session is defined by multiple interactions and some of these interactions can be dependent on previous events. This necessitated the development of systems which also monitored the responses out of the SUT and dynamically varied the delivered data based on these responses. This led to the construction of dynamic content generation in fuzz generators.

One example of a fuzzer that uses a non-standard delivery mechanism is the configuration fuzzer (Dai, Murphy & Kaiser 2010). This fuzzer is designed to randomly alter the runtime configuration of the SUT while processing inputs and uses a delivery mechanism that directly modifies the process's memory. Delivery mechanisms like direct memory injection may seem conceptually simple, but actually require significant complexity to avoid corruption of the SUT and thus spurious errors.

Remote fuzzing is primarily done across the network interface, although monitoring of the results can be more difficult because of the the lack of access to the SUT for observation. Use of remote SQL and HTML queries can sometimes reveal information about the internal state of the remote system due to subtle differences returned by the SUT[5].

5.3 Classical Monitoring Systems

The ability to detect that system input has resulted in an error is critical to the fuzzing process. In addition to detecting errors, the monitoring system often cooperates with the fuzz generator to determine which input sets contributed to the flaw being expressed. There are two broad classes of monitoring system. Local monitoring systems are used when the system under test is installed and executed on the same system as the monitoring system. Remote monitoring systems are used where the monitoring system can only interact with the system under test by monitoring its input and output behaviour.

Local Monitoring Systems Initially local monitoring systems were simple and rudimentary. In the initial UW-Maddison study the monitoring process consisted of launching the SUT, feeding it the input set and waiting until the process terminated. After the process had terminated the file system was checked for the presence of core dump files. The presence of a core dump signaled that the system had failed, whereas the lack of such files indicated that no errors had been found. Local monitoring system have increased in sophistication since that time and now often use an augmented run-time environment in order to improve error detection. On example is the Microsoft Application Verifier, which monitors the application's interaction with Windows APIs in order to detect potentially unsafe usage patters and to detect any raised exceptions (even if they are handled.) Another example is offered by Muniz & Ortega (2011) who, in order to fuzz Cisco's IOS

[5]Blind SQL injection uses varying Web error messages to determine the effects of injecting SQL code into a database system. The output of this system is not otherwise accessible to the user

operating system, loaded the OS image into a simulator which had been modified so that GDB and IDA Pro could be used to monitor IOS's execution.

Remote Monitoring Systems Given the limited ability of a remote monitoring system to observe the SUT, they have historically been less capable than their local counterparts. Early examples typically looked for interruptions to network communications, signified by either a TCP reset packet or a timeout for either a TCP or a UDP communication channel. Given the increasing complexity of network based applications, it was unsurprising that more advanced remote error detection methods have become available. For example, if a web-based application is running as a Tomcat servlet then an attached monitor can also examine the output from the Tomcat server to detect and analyse errors within the servlet.

5.4 Modern Fuzzers

Despite their simplicity, classical/black-box fuzzing techniques were powerful and often effective. SPIKE is a good example of this: its internal structure is basic, but its data handling abilities are powerful and expressive.

Once basic fuzzer functionality stabilised, designers turned their attention to making fuzzers more efficient. Their primary approach was to restrict the fuzzer's search space to data items which were of interest. For security fuzzers, these were data items that could affect security critical parameters e.g. size fields for buffers created by system calls like `malloc()` and `strncmp()`.

In search of an effective technique to determine these parameters, fuzzer writers decided to relax the black-box assumption and look inside the box. Any fuzzer which does not use the black-box assumption we classify as a *Modern Fuzzer*.

As many fuzzer writers did not have source code for their SUTs, they turned to reverse engineering tools – debuggers, disassemblers, and static structural analysers to map the SUT's code execution graph – These tools allowed analysis of the SUT to identify some items (such as calls to libraries) which might be security critical and vulnerable, but also allowed the writers to track the flow of data through the program. This led to data based analysis techniques such as symbolic execution and dynamic taint analysis.

A modern fuzzer may use both generative and mutative methods and also hybrids of the two in its fuzz generator. The Evolutionary Fuzzing System (EFS) (DeMott, Enbody & Punch 2007) is notable for using a Genetic Algorithm for mutating data (from an initial randomly generated set).

A modern fuzzer usually uses similar delivery mechanisms to a classical fuzzer, however it might also use memory based delivery. With a memory based delivery mechanism the fuzzer is able to directly modify the memory contents of the target system. Among other things this can allow the fuzzer to bypass parts of the system's input processing, for example bypassing check-sum verification routines by presenting fuzz data to functions that are normally protected by a check-sum verification step. Another example is the Dai, Murphy & Kaiser (2010) configuration fuzzer, which randomises the system under test's configuration with each test case (Dai, Murphy & Kaiser 2010).

Modern (Grey) fuzzers can have more sophisticated target monitoring than classical types. Classical White-box fuzzers can have full (traditional) debugging abilities (since they have source code and compiler symbols available to them). Classical black-box fuzzers do not typically monitor the internal program state of the SUT but examine the system after it has crashed/failed. Grey-box fuzzing can use hybrid techniques such as a *Runtime Analysis Engine* which monitors the System-Under-Test (SUT) but is capable of determining finer grained data about the SUT runtime environment and execution state instead of just "running vs crashed". Some Runtime Analysis Engines are capable of comprehensively instrumenting the SUT code, including replacing instructions and data symbols with their own library calls and symbols.

Taint analysis tags the data input into the SUT and subsequently analyses the SUT either during the execution flow or post a failure to examine how the program used the input data and which program elements were touched ("tainted") by the data. In this way the search space can be constrained, as any data which does not taint a program region of interest can be removed from the fuzz set.

Associated with data tagging, Symbolic Execution instruments an SUT with symbols replacing "concrete" (static input dependent) values in the program. When these symbols are executed, a Path Constraint is constructed to indicate which branch was taken due to the symbol. Eventually a Constraint Solver can determine the conditions of the branch and use this information to adjust the data, allowing the other part of the branch to be explored. These techniques are used by Microsoft's SAGE fuzzer (Godefroid 2010), which reportedly found a third of the Windows 7 bugs between 2007-2009.

Use of library hooking and dynamic loading techniques are used by GuardMalloc and Valgrind (Nethercote & Seward 2007). These two systems monitor every malloc and free made by the SUT and Valgrind uses a synthetic processor to monitor memory accesses. Use of virtualisation technologies is becoming more common as it becomes more widespread and cheaper.

These are sophisticated techniques which establish a feedback path between the target monitoring code and the Fuzz Generator. However, some of these techniques are quite slow and require off-line processing. Nonetheless they are remarkably powerful and are many times more efficient than random fuzzing, especially when coupled with the search for a vulnerable goal state found by structural analysis. These techniques are referred to as *directed fuzzing*.

Use of fuzzing techniques by the software quality community also brought with it different monitoring systems which were not available or useful to the black hat community. These included development of *Reporting Engines*, which reported the results of the fuzzing to developers to allow the vulnerability to be fixed. These were capable of automatically generating documents in required bug-report formats. Also, as part of the automated test system for white-box fuzzing (but also model-based black-box fuzzing), *Oracles* were developed to provide the "right" answer for a given data input. Grey-box fuzzers can also use these techniques by extracting code fragments from the SUT in order to construct an Oracle (e.g. Taintscope takes check-sum generation execution traces from the SUT and solves them to generate correct check-sums that the SUT will accept).

Takanen (2009*b*) discusses the trend for more sophisticated data representation within fuzzers (both classical and modern), especially fuzzers which use model/protocol description approaches. Initial black-box fuzzing efforts were primarily based around ASCII; ASN.1 became necessary as protocol generators became popular (and is used in Codenomicon), and XML is used in Codenomicon, Peach and MuDynamics to represent complex structures.

6 Experiences

After completing the basic literature review, a small, representative set of fuzzers was chosen for use in conducting some basic experiments. Before conducting these, a small collection of programs was written, with each deliberately containing a common basic flaw, like buffer overflows or NULL pointer dereferencing. These toy programs were intended for use in establishing a baseline for the selected fuzzers. Initial expectations were that these fuzzers and fuzzing frameworks would be easy to use and, in the case of frameworks, adaptable for the basic test cases. This, however proved not to be the case.

The first problem was that many of the tools were discovered to be immature, unstable, or poorly supported. For example, having downloaded `fuzzgrind's` source code, significant levels of investigation and modification were required before it would compile. After addressing its compilation problems we attempted to use `fuzzgrind` to fuzz the collection of toy programs, only to discover that, for reasons that remain unclear, it was incapable of uncovering any of the flaws they contained. The lack of online information or support for `fuzzgrind` meant that, despite further efforts to diagnose the problem, we were unable to establish even baseline functionality. Experiments aiming to use JPF were aborted even earlier due to its dependencies on a large number of libraries that are no longer supported.

Another set of experiments was conducted using `klee`. Unlike `fuzzgrind`, `klee` is still undergoing active research (as of 2011). However this, combined with the fact that it is built on top of `llvm` (a rapidly evolving compiler architecture), meant that there were still some issues ensuring that the versions of the `klee` and `llvm's` source were compatible. Once it was compiled `klee` was used to fuzz the test programs. The first step of this process was to compile the programs into `llvm's` byte code representation, with the resulting binaries given as input to `klee`. In each case `klee` successfully identified all of the possible code paths through the toy programs, provided examples of input that would result in each code path being executed, and identified which paths represented flaws.

After establishing that `klee` could handle the basic test cases, attention was turned to more complex examples. Two systems were selected as targets for this round of fuzzing. The first was an open source PDF file viewer called MuPDF[6]. The second was a trusted kernel developed as part of the Annex project(Grove et al. 2007). In both cases however, problems were encountered while attempting to build an `llvm` bytecode version of the programs. This was due to inherent differences[7] in the build processes used by `llvm` and `gcc`, with both projects designed to build using the latter. Unfortunately, in both cases, the effort required to remedy the situation was deemed too great to proceed further.

The Peach framework proved to be a useful environment for fuzzing however it is far from being a mature tool and required a lot of effort to achieve real-world goals. Peach required significant effort to install and run, with many dependencies on both the Windows and Linux platforms. Once installed, there were several bugs in the current (v2.3.8) release which limited or degraded functionality. Some of these were fixed in the latest source which needed to be separately downloaded and patched. For example, there was a bug with the Agent on Windows where it could not monitor CPU usage of the target

[6]http://www.mupdf.com

[7]In `gcc`, the compiler can also control the linking process; this is not the case with `llvm`

application correctly. Due to this problem, Peach simply launched the application for 5 seconds and if there was not a crash, it forcibly terminated the application, assuming no faults were found. A patch for the Peach source fixed this problem by monitoring the CPU usage of the target. If CPU usage dropped below a certain threshold for a certain period, Peach assumed that the target had finished processing the input and forcibly terminated it. This was not an effective solution as applications often detected a problem with the input document and prompted the user for action before continuing to load it. This process was interrupted by the forcible kill. There was some functionality in Peach to detect pop ups of the target application and acknowledge them, however it was not mature enough to work in most cases.

As Peach was developed in python, it was possible to work around these problems by fixing and enhancing the source. By doing this, it was possible to achieve some meaningful results using Peach as a starting point.

A common characteristic of the three commercial fuzzers discussed in Appendix B is the possession of sizable protocol libraries, especially of network protocols. The availability of these libraries is one characteristic that sets the commercial fuzzers apart from the open-source ones (with support being another); the very high cost of the software and services reflect this utility advantage.

7 Conclusions

7.1 The State of the Art

Traditionally fuzzing has been black-box whereas static analysis has been white-box. Both areas are trending towards grey-box as a result of the increasing availability of tools to support binary analysis. The techniques used to generate semi-valid data inputs are becoming more sophisticated. These include the use of genetic algorithms or context free grammars instead of static techniques such as templates). Modern fuzzers are able to understand better the internal characteristics of programs, as they can apply techniques such as the static analysis of functions with a binary and the tracking of the code execution graph within a running program. Whilst early fuzzers were unsophisticated with their choice of data, a modern fuzzer restricts the search space of the data it presents, eschewing invalid data (such as data with incorrect checksums) for semi-valid data, and also favours cases to test boundary conditions. As a result, a modern fuzzer may be many times more computationally efficient than a classical fuzzer.

The last ten years have also seen the proliferation of open source fuzzing tools and related frameworks. Our experience found that many of these were immature or hard to use. Recent times have also seen the adoption of fuzz-based software testing in many commercial software teams in turn giving rise to commercially supported fuzzing products. These trends seem likely to continue and will reduce the effort required to develop a fuzzer for new systems. The Fuzzing email list shows that the trend for the last five years is toward generators over mutators.

7.2 Fuzzing in the Future

The commercial fuzzers show a trend towards use of protocol modellers, with increasing numbers of protocols being available in their libraries. It appears that this is still a fruitful avenue for fuzzer development, and it is expected that this effort will continue. It is difficult for open source fuzzers to compete with the commercial fuzzers in this respect, as construction of a comprehensive protocol library within a single framework is better matched to a paid effort than a volunteer one. Any open-source analog of this effort is likely to come from the Metasploit community (which is a general purpose framework which supports fuzzing modules, rather than a fuzzing framework per se).

We forecast a continued trend towards grey-box fuzzing, including improvements in fuzzer's ability to affect the SUT and to construct feedback loops between the monitoring system and fuzz generator components. The use of symbolic execution techniques and code execution graph analysis, logically coupled with even basic AI techniques (such as graph based searching) are likely to lead to the construction of an automated target exploration/exploitation toolkit in the next half decade. Acceptance of fuzzing as a white-box technique in the software quality community (along with negative testing generally) is expected to grow.

Use of virtualisation technology is expected to grow - not only for massively parallel/distributed fuzzing (using Cloud resources), but also for instrumentation of virtual processors and for the use of snapshots to rapidly roll back machine state and compare between machines. Utilising cloud computing to conduct fuzzing is expected to be a cheap method of deploying a very very large number of fuzz machines for a short period of time. A distributed monitoring and reporting system would be required for this endeavour.

One possible area under-explored by previous fuzzers is the notion of time. Although some protocol based fuzzers may include some knowledge of time-based state changes, it is possible that fuzzing the timing of input data may cause race conditions that uncover many instances of program failure, especially as programming models trend towards greater use of parallelism and distributed computation.

Classical fuzzers relied on program failure to determine fuzz success. Grey-box techniques allowed instrumentation of the execution flow of the SUT. We expect that the monitoring systems will be improved to detect other classes of program flaws, i.e. not just buffer overflows and memory allocation problems, but also logical (branch) errors, protocol flaws (when coupled with a good protocol modeller), and security violations such as granting unauthorised access.

Appendix A Literature Survey

Automatic Generation of Test Cases (Hanford, 1970)

Hanford describes the Syntax Machine, a tool developed to automatically test the syntax and lexical analysis components of a compiler. Given a definition of the grammar accepted by the compiler, the Syntax Machine can generate three classes of source code suitable for evaluating different aspects of the compiler's behaviour. In its standard mode of operation the Syntax Machine generates random programs that are well formed with respect to syntactic and lexical requirements of the language. In another mode the Syntax Machine is also able to systematically generate programs with all forms of specific components of the grammar. The final operational mode deliberately incorporates syntactical and lexical errors into the generated code. In all cases, the fact that the correct result for the program is unknown prevents the Syntax Machine from being able to detect errors in later stages of the compilation process.

The grammar accepted by the Syntax Machine is based on a Backus-Naur Form (BNF) context free grammar (Backus 1959). As most programming languages are not context-free, the BNF is extended to create what the author calls a dynamic grammar. A dynamic grammar extends the BNF in two important ways. The first extension allows production rules to be added dynamically to the grammar that the Syntax Machine is given. The second extension allows for $not - rules$ which are used to add constraints that can block specific productions just before they are committed. An example of where both of these rules can be applied is with the rule for producing a variable declaration. Each time the rule is used, it adds a production rule into the grammar such that the newly declared variable can be produced wherever a variable name is required. Further, declaration rules can also use a not-rule to prevent re-declaration of a variable with a name that has already been used.

An Empirical Study of the Reliability of UNIX Tools (Miller, Fredriksen & So, 1990)

While connected to their university workstation via a dial-up line, one of the authors noticed that line noise was generating frequent spurious characters, often including characters not generated by keyboard input. Further, these characters were interfering with his ability to use the system, not only adding garbage characters to his input, but also crashing many of the UNIX programs he was attempting to use. This scenario inspired an audit, looking for similar flaws in various UNIX utilities on a variety of UNIX flavours. The tests covered two kinds of utility, programs that accept input from `stdin` and interactive programs that accept input from a console. After completing the audit it was found that in six versions of UNIX, between 24.5% and 33.3% of the utilities tested failed at least one test.

Tests were performed using two tools, `fuzz-generator` and `ptyjig`. `fuzz-generator` creates an unstructured stream of random bytes, writing them to `stdout`. The fuzz generator has an option to include non-printable characters in its output, and another option to include zero bytes. Testing an application that reads from `stdin` is as simple as piping the output from the `fuzz-generator` into the program. Testing an interactive program, on the other hand, requires the use of `ptyjig` to create a pseudo-terminal, to which the target utility can attach. Input from the `fuzz-generator` is then fed to `ptyjig` which turns the input into random key-strokes from the psuedo-terminal.

Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services (Miller, Koski, Pheow, Maganty, Murthy, Natarajan & Steidl, 1995)

Five years after the publication of (Miller, Fredriksen & So 1990) the authors re-evaluated the reliability of a number of core UNIX services. The original paper discussed the use of random input to test the robustness of command line and console based utilities. Tests in this paper were extended to include the handling of failed memory allocation calls, as well as to UNIX's network services and X-Windows applications and servers. Despite the free availability of the fuzz testing tools and advisories sent to the software vendors the authors found that many of the flaws uncovered in the original paper remained unpatched. While many X-Applications were found to contain errors, no flaws were discovered in any of the network services or X-Servers tested.

The first round of console testing was again performed using *the fuzz generator* and *ptyjig* programs from the original paper (see above). Testing of networking services was made possible through the use of a new utility, called portjig, which established a socket connection to the service over which it forwarded input provided by the fuzz generator. Another new tool, *xwinjig*, was developed to interpose between an X-Windows server and an X-Windows application. When testing X-Windows applications *xwinjg* was used to generate four different kinds of random data stream. Type one data streams consist of purely random data. Type two streams distort legitimate messages sent between the server and application. Type three streams randomly injected well formed events into communications from the server to the application with random sequence numbers, time-stamps and payloads. Type four streams are similar to type three streams except that the events injected into the stream correctly account for details like sequence numbers and window geometry, making them logically indistinguishable from legitimate events. The final tool developed was *libjig*, which could be used to intercept calls to functions within the C standard library. In this case *libjig* was used to intercept calls to the memory allocation function *malloc*, at which point it would randomly either simulate an allocation failure by returning *NULL* or pass control to the original function and return the results.

An Empirical Study of the Robustness of Windows NT Applications Using Random Testing (Forrester & Miller, 2000)

The third publication related to the fuzzing work at UW-Maddison applies the fuzzing approach to testing a suite of Windows applications. Much like X-Windows, the application inputs consist of a series of events, to which the application is expected to respond. Two broad classes of events were identified for fuzzing purposes, Win32 messages and system

Table 1: *Windows NT Fuzzing Failure Rates*

	SendMessage	PostMessge	Keyboard & Mouse
Crashes	24 (72.7%)	30 (90.9%)	7 (21.2%)
Hangs	3 (9.0%)	2 (6.0%)	8 (24.2%)
Total	27 (81.7%)	32 (96.9%)	15 (45.4%)

events. A tool, once again called fuzz, was developed capable of generating random events from these two classes and used to test the application suite.

Three kinds of fuzz testing were used to test the application suite. Both the first and second testing scenarios sent randomly generated Win32 Messages to the target application. These messages combined a random valid message type with randomly generated values for the message parameters. When a Win32 Message is given to an application it can be done using either the asynchronous `SendMessage` function, which was used in the first testing scenario, or via the synchronous `PostMessage` function which was used for the second scenario. The third testing scenario hooks into the system event queue to mimic random keyboard and mouse events.

Once again fuzz testing was sufficient to yield impressive results. In total thirty-three applications were tested under each of the scenarios, none of which was found to be robust under all of the three testing scenarios. The number of applications that crashed and hung are summarised in the table below.

Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting (Sparks, Embleton, Cunningham & Zou, 2007)

The approach described in this paper uses the combination of a partially defined input structure and a genetic algorithm with an evolutionary heuristic that is intended to derive inputs that will result in a control flow that reaches a target code segment. The approach is designed to improve on both the code coverage and depth of analysis in comparison with a random fuzzer.

Analysis began with a tester supplied Context Free Grammar which was used for input construction, and the extraction of the control flow graph from the target's binary image using a tool like IDA Pro[8]. The tester then identified one of the nodes within the control flow graph as the target node. The graph was then simplified, collapsing all transitions to a state from which the target node could not be reached into a single transition to a generic rejection state. Further, given that once the target state had been reached any further transitions could be ignored, all transitions from the target node are replaced with transitions to back to the target node. The simplified graph was then used to generate an absorbing Markov Model in which the transition probabilities corresponded to the likelihood of the transition based on random input. The PaiMei framework[9] was then

[8]http://www.hex-rays.com/products/ida/index.shtml
[9]http://www.openrce.org/downloads/details/208/PaiMei-Reverse-Engineering-Framework

used to monitor the execution of the target program as it processed input, yielding the sequence of state transitions made. Once a generation of inputs have been processed, the probabilities within the Markov Model were updated and the fitness of each input was calculated. The fitness value was defined as the inverse of the product of the probabilities for the transitions within the model, and was intended to favour inputs that exercised less common transitions within the control flow graph. Once the inputs used in a given round had been ranked, the evolution step used single point cross over, elitism, insertion, deletion and point mutations to generate the next round of inputs. However, the evolution process was not directly applied to the inputs. Instead, because the inputs were generated by a grammar it was possible to represent any input with a sequence of numbers, where the n'th number corresponded to the production rule applied in the n'th step of generating the input. The evolutionary process targeted these number sequences instead of the inputs themselves, an approach known as Grammatical Evolution (Ryan, Collins and O'Neill 1998).

Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing (DeMott, Enbody & Punch, 2007)

The Evolutionary Fuzzing System (EFS) is designed to learn the input protocol that the target interface uses while fuzzing that interface. EFS's ability to learn protocols is based on the use of a genetic programming approach that attempts to evolve a collection of communication sessions which combine to maximise code coverage within the tested system. The authors propose that one benefit of an evolution based approach is that it can automatically test for vulnerabilities at different depths within the system, proceeding deeper and deeper into the system as it better learns the protocol.

EFS is built using both the General Purpose Fuzzer (GPF) and the PaiMei reverse engineering framework. The PaiMei framework is used to analyse the target software's binary code to locate functions and basic blocks and to attach to the running process and use breakpoints to monitor execution flow. GPF is then used to randomly generate a set of initial sessions, possibly seeded with some limited a priori knowledge of the protocol. The sessions are then grouped together into a set of pools and the first round of evolution starts. Each round of evolution begins by presenting each of the sessions within the pools to the target program for processing, yielding a report of the resultant execution flow from PaiMei. The reported execution flows are then used by the fitness functions in both intra-pool evolution, which assigns higher fitness values to sessions with an execution flow that includes more basic blocks, and inter-pool evolution process, which considers pools as a collection of basic blocks touched and prefers those pools that collectively touch more basic blocks that are not touched by any other pool. In both evolutionary steps elitism is used to preserve the top ranked session / pool respectively. The remaining candidates are then run through a cross over process, which favours genes from the higher ranked candidates, and also run through a mutation process.

A July 2009 review of the system (Alverez 2009) states that "the code coverage approach is extremely poor and the pathflows approach is very time consuming. It heavily depends on the implementation how effective it will be in the shortest amount of time."

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (Cadar, Dunbar & Engler, 2008)

The authors discuss a new tool called KLEE which uses symbolic execution to analyse programs and automatically generate system input sets that achieve high levels of code coverage. KLEE is specifically designed to support the testing of applications that interact with their runtime environment. KLEE was used to test the GNU Coreutils suite of applications, which form the basis of the user environment on many different Unix like systems. Similar tests were also completed for the BusyBox suite of applications for embedded environments. Finally KLEE was used to test the HiStar kernel. KLEE's symbolic execution engine accepts programs that have been compiled to Low Level Virtual Machine (LLVM) byte code which it then symbolically executes with two goals. First, it attempts to touch every line of executable code in the program. Second, at each potentially dangerous operation, such as memory dereferencing, if any of the possible input values can cause an error.

Automated Whitebox Fuzz Testing (Godefroid, Levin, Molnar et al., 2008)

The Scalable, Automated, Guided Execution (SAGE) fuzzer uses a novel algorithm, called generational search, to generate sets of system input in a way that is intended to improve the fuzzing of applications that accept highly structured input like compilers and interpreters with large input files. Under generational search a symbolic execution engine is used to collect the path constraints for the execution flow followed when processing a seed input file. These constraints are then used to identify the next generation of potential inputs. For each constraint in the path the constraint is negated and combined with the constraints that occur before it. If the resulting set of constraints remains solvable, a new system input set is generated. This input set is used to test the SUT. In addition to testing the SUT each test case is also scored based on the increase in code coverage it achieved, and stored in a pool of potential seeds files. After all of the input sets for a set of path constraints have been generated, the candidate with the highest score is selected from the pool of potential seeds and the process begins again using that file as the seed.

SAGE discovered numerous vulnerabilities, including a remote execution vulnerability in the processing of cursor animation files that had been originally been missed despite the extensive in house use of fuzzers with knowledge of the .ANI file format(Howard 2007). Over two ten hour sessions, each seeded with a different input file, SAGE found 43 crashes within Microsoft Office 2007.

Grammar-based Whitebox Testing (Godefroid, Kiezun & Levin, 2008)

Experience revealed that the effectiveness of the SAGE fuzzer (see above) was limited with respect to testing applications that process highly structured inputs. Such applications

Table 2: *Effectiveness of Different Fuzzer Types*

strategy	number of tests	% total coverage	% lexer reach	% lexer coverage	% parser reach	% parser coverage	% code generator reach	% code generator coverage
black-box	8658	14.2	99.6	24.6	99.6	24.8	17.6	52.1
grammar-based black-box	7837	11.9	100	22.1	100	24.1	72.2	61.2
white-box	6883	14.7	99.2	25.8	99.2	28.8	16.5	53.5
white-box + tokens	3086	16.4	100	35.4	100	39.2	15.5	53.0
grammar-based white-box	2378	20.0	100	24.8	100	42.4	80.7	81.5

typically process input in stages, for example the lexer and parser stages of a compiler, and SAGE's analysis essentially became trapped exploring the vast number of control flows in these early stages. In an attempt to facilitate the fuzzing of deeper components of such systems SAGE was extended to incorporate knowledge of the grammar accepted by the system. This knowledge was used in two ways. Firstly, the satisfiability test used to validate potential system input sets was extended to incorporate grammar based constraints along with the path based constraints derived from symbolic execution. Secondly, knowledge of the grammar allowed the fuzzer to generate lexically valid tokens. Both of these enhancements allowed the significant levels of pruning to occur with in the early stages of the search tree, thus facilitating the testing of deeper regions of the system.

The resulting system was used to test the JavaScript interpreter in Internet Explorer 7 (IE7). Rather than discussing any vulnerabilities that were found the analysis of the effectiveness of the approach focused on the relative levels of coverage and depth reached in relation to other fuzzing approaches. Several fundamental approaches were used to fuzz the IE7's JavaScript engine for two hours and the results, which are reproduced in the table below, show that the grammar-based white-box approach yielded the best results in both total coverage and reach.

Experiences with Model Inference Assisted Fuzzing (Viide, Helin, Laakso, Pietikäinen, Seppänen, Halunen, Puuperä & Röning, 2008)

The paper's authors start with the hypothesis that, to efficiently test a program the input test data must have a structure that is approximately correct. One existing approach to achieving this is to randomly mutate existing valid inputs. However, without an understanding of the data's structure, the level of mutation can be severely limited. The idea of model inference assisted fuzzing is intended to address this problem by inferring the data's structure from a set of training inputs.

Rather than adopting one of several existing systems for structural inference (Beddoe 2005, Larsson & Moffat 1999, Cui, Kannan & Wang 2007, Lehman & Shelat 2002) the author's opted to construct their own. Initial work started with a structure definition language that, whilst flexible and powerful, was not suitable for a lightweight inference process. The authors found that using a subset of the language's expressive power, equivalent to that of a context free grammar, offered a good balance between expressiveness and ease of inference.

To test the approach the authors attempted to infer a grammar for ten different file compression formats[10]. The resulting grammars were then fed into a reverse parser to generate a collection of test inputs. In addition to the fuzzing inherent in the structure inference process, the reverse parser also introduces random mutations to the grammar as it is used. The resulting files were then used to test five anti-virus programs, finding multiple flaws in four of the five programs.

Taint-based Directed Whitebox Fuzzing (Ganesh, Leek & Rinard, 2009)

The BuzzFuzz tool described in this paper is a file fuzzing tool that uses dynamic taint-tracing (Newsome and Song 2005), a set of key points of interest within the program, and a pre-existing collection of legitimate input files to drive a fuzzing process that targets those points. The fuzzing process uses the results of the dynamic taint analysis process in an attempt to determine which regions of each legitimate file affect the values used at the program's key points so that it can focus fuzzing efforts on those regions of the file. The main intended benefit of this approach is an ability to fuzz the deeper "core-logic" of the system as compared to the file parsing logic which is normally tested by a fuzzer.

Given the source code for the system being tested, the set of key points and collection of input files, BuzzFuzz starts by annotating the source code with calls to the BuzzFuzz library. These calls allow BuzzFuzz to track for each calculation which input bytes affected the result along with the type information. The instrumented version of the program is then run on each of the pre-existing input files, which generates a set of reports listing the input bytes and their types that influence the values used at each key point. Each input file and its associated taint trace report are then fed into the directed fuzzer to generate fuzzed output files, which are then run through the non-instrumented version of the program for testing.

Configuration Fuzzing for Software Vulnerability Detection (Dai, Murphy & Kaiser, 2010)

This paper describes a system that fuzzes the run-time configuration of a program. The approach is based on the premise that different system states can expose a different subset of implementation flaws within a system. The resulting fuzzer builds on the concepts of In Vivo Software Testing (Murphy, Kaiser, Vo and Chu 2009), which support continued testing of a system even when deployed. One idea behind the use of an In Vivo approach is that it allows the system to be tested against the diverse range of real world inputs without requiring the development of a second fuzzer to generate the input with which to fuzz the system.

[10] `ace`, `arj`, `bz2`, `cab`, `gz`, `lha`, `rar`, `tar`, `zip`, and `zoo`

The implementation described in the paper uses a white-box approach that starts with the tester identifying a set of configuration variables and settings within the system's source code. Once identified this information is used to generate a fuzz-config function, which when called randomises the system's configuration. In the next step the tester annotates the selection of functions they have chosen as the instrumentation points for the configuration fuzzing process. For each annotated function the pre-processor emits three functions. The first is simply the original function renamed to -function. The second is a wrapper function with the name of the original function. The wrapper function merely calls `fork()` to duplicate the process. In the parent process the function is called, while in the child process control is passed to the third function generated by the pre-processor test-function. The code generated for test-function is another wrapper where the call to the original function (now renamed) is preceded by a call to `fuzz-config()` and followed by a call to `test-invariants()`. The `test-invariants()` function, which is supplied by the tester, checks for violations of any invariants that should hold, including security properties.

TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection (Wang, Wei, Gu & Zou, 2010)

This paper continues the work by a number of authors on dynamic taint analysis. Three of the four authors are based in China and the work is funded by a grant from the Chinese NDRC. Taintscope is a modern fuzzer, employing symbolic execution and taint analysis, however it is a proof-of-concept rather than a general purpose fuzzer at this point, targeting code that generates checksums. Nothing would prevent this work from being generalised. The prototype Taintscope is a file based fuzzer.

The program has its own debugger ("Execution Monitor") and instruments the SUT, examining each conditional jump instruction. It uses a simple heuristic to identify potential checksum code and follows it with deeper analysis to confirm this hypothesis. Data input into the system is tagged to identify "hot bytes" (which are definable security relevant bytes such as length fields in malloc and strcpy calls). Taintscope's fuzzing effort is directed at these hot bytes, and contains a number of "attack heuristics" for these hot bytes. Data protected by checksums can have the checksum code disabled by rewriting the branch instructions, although Taintscope also records the execution traces of the checksum algorithm, regenerates the relevant checksum data and writes them into the malformed input file for subsequent replay. Checksum data is handled as symbolic data; for speed most of the data input file is treated as concrete data.

Unlike BuzzFuzz, Taintscope does not need access to the source code to perform its taint analysis. It is a grey/black-box tool.

Finding and Understanding Bugs in C Compilers (Yang, Chen, Eide & Regehr, 2011)

This paper describes the development of, and experiences using, Csmith. Csmith is a randomised test generator designed to improve the quality of C compilers. One of the key design considerations in the development of Csmith was the desire to focus testing on the deeper components of the compiler, with an emphasis on the logic for code generation and optimisation. Satisfying this goal both placed stringent constraints on the generation of input sets and also required the use of non-standard techniques for flaw detection.

Each test set generated by Csmith is a randomly generated C source file that computes a single value `checksum`, which by default is printed as the final step in `main()`. Prior to testing, the correct value for `checksum` is unknown, so the testing process can't validate the result using a priori knowledge. Instead, the same test set is compiled with multiple compilers, and the results obtained by executing resulting binaries are compared. If the results are the same, then the compilers are deemed to have worked correctly. If more than one result is given then at least one of the compilers contains a flaw.

In order to ensure that there can only be a single valid result Csmith takes care to generate C source code that is free of both undefined and platform dependent behaviour. One example of undefined behaviour is that the result of performing bitwise operations on signed integers isn't defined within the C standard. The differences in the number of bytes used to represent the `int` type is an example of platform dependent behaviour. In total one hundred and fourteen types of such behaviour were identified that needed to be avoided.

Csmith's code generation process doesn't take any steps to ensure that the resulting program will terminate in the the time allotted for testing. The end result is that approximately ten percent of the programs generated apparently don't terminate. In the case of different behaviours with respect to program termination it can't be conclusively determined that an error has been found, but it is a useful precursor to suggest further investigation.

Using Csmith over a span of three years Yang, Chen, Eide & Regehr (2011) were able to find errors in each of the compilers tested, including both open source and commercial products. In total three hundred and twenty-five (325) previously unknown compiler bugs were reported to vendors, mostly in the middle end code generation and optimisation stages.

AEG: Automatic Exploit Generation (Avgerinos, Cha, Hao & Brumley, 2011)

Other tools have focused solely on the discovery of errors, This work attempts to discover which flaws are exploitable with respect to control flow hijacking and to craft an exploit that creates a shell.

Appendix B Tools & Frameworks

Table 3: *Table of Fuzzing Tools and Frameworks*

Product Name	White/Grey/Black	Generation or Mutation	Target Interface	Failure Detection	Comments
Codenomicon DEFENSICS (commercial)	Black, White & Grey modes available	Generation from Test Cases and protocol plugins, mutation based on packet captures	Network primarily but also Files	Debugger and Oracle	Commercial Product, has vulnerability library and plugins for 200+ protocols
BeStorm (commercial)	Black-box	Generation, via protocol specifiers	Network	Debugger; also has a monitoring component for SUT	
MuDynamics (commercial)	Black-box	Generation, via protocol modules; Mutation from network captures	Network only	Not clear. Seems to be able to interroperate with third party test equipment (e.g. Agilent)	Uses XML templats and claims to be able to fuzz anything expressible in XML.
SPIKE	Black-box	Generation from Templates and rules	Network (SpikeFile for files)	None - provide your own debugger	One of the early Black Hat fuzzers. Simple and powerfull. Allowed data to be handled as blocks
The Art of Fuzzing	Black-box	Mutation from pcap data fields	Network	None - provide your own debugger	GUI interface made network fuzzing accessible. Now Defunt
General Purpose Fuzzer	Black-box	Mutation of network pcap	Network, also File	Use of debugger	Notable for spawning EFS. Mutation can be quite smart due to packet parsing & tokenizing / protocol analysis
Evolutionary Fuzzing System	Greybox, when coupled with PaiMai reverse engineering tool	Mutation	File	As for GPF	Uses Genetic Algorithm for data generation, static analyser for binary analysis/coverage determination and identification of vulnerable structures. Uses code coverage tool
Peach	Blackbox	Generation and Mutation from XML protocol descriptions	Network and File	Monitor agent and WindowsDebugEngine	Remote Monitoring Agent monitors for crashes. Can resume search space from/around crashes. Has code coverage tool. XML syntax has learning curve. Is capable of Constraint based fuzzing
Sulley	Blackbox	Generation and Random Mutations	Network and File	Crash monitor and Post Mortem analysis	Evolutionary fuzzer which takes successful features from previous systems. Has debugger monitor, crash restore, data block handling, attack library etc
Flashboom	Blackbox	Generator	Invocation Parameters via Browser/Flash	Execution watchdog & XML network socket (status)	Fuzzes only a specific native interface function in Adobe Flash. A specific tool, not a framework
Spider Pig	Blackbox	Generator	File and Invocation Parameters	Debugger	Fuzzes Javascript used within PDF files. Brute force random generator. Single purpose tool not extensible

B.1 Codenomicon DEFENSICS (2001–present)

Codenomicon[11] can use pre-defined test cases. It also has a vulnerability library and the ability to generate semi-valid cases based on pcap network capture. It also has "mini-simulation" support, which allows data generation based on Java rules written to describe both the data semantics and the communication (syntax and behaviour) between entities. Codenomicon is a commercial product and claims to have protocol support for 200+ network protocols. It is a commercialisation of the PROTOS program (from the Secure Programming Group, University of Oulu Finland).

B.2 beStorm and Mu Dynamics

These two are commercial fuzzers. They are both black fuzzers of network protocols, and have the ability to generate network traffic using protocol modules. Mu Dynamics[12] is XML based and can generate traffic based on XML specifications. Both systems provide some information about their fuzz generator, but lack technical information regarding the rest of the system. It is not possible to comment further about the capability claims in their marketing literature.

B.3 SPIKE (2001–present)

SPIKE [13] is a framework which allows a user to specify an expected protocol using simple templates. It provides an API for generation of traffic against those templates. For example an expected transaction could be specified as fields and data blocks, and the fields could be incremented and iterated. The framework provided basic socket and memory allocation support. It was very popular because of the simplicity of its internal representation; this simplicity still allowed a great deal of expressiveness. It is an early example of a brute-force generation based fuzzer.

B.4 Valgrind (2004–present)

Valgrind runs the SUT in a synthetic processor and monitors every memory allocation, deallocation and memory access. It hooks the dynamic loader and replaces the malloc library call with accesses to itself. (Nethercote & Seward 2007)

B.5 The Art Of Fuzzing (TAOF) (2006–2009)

The Art of Fuzzing [14] was a Python based generation fuzzer used primarily for network transactions. It provided a simple GUI interface and the specification of fields and data types within a network capture. It is now defunct, with no activity since 2009. The code has been abandoned in Sourceforge and the website is offline.

[11]http://www.codenomicon.com/
[12]http://www.mudynamics.com
[13]http://www.immunitysec.com/resources-papers.shtml
[14]http://sourceforge.net/projects/taof/

B.6 Peach (2006–present)

Peach[15] is a modern cross platform fuzzer which performs both generation and mutation. The user generates a series of XML files (called "Pits") which describes the protocol and how to run the test. (A number of pre-written pits are available). Each pit can take in user supplied data blocks as a starting point and can specify fields to be fuzzed for particular file types. A number of "transformers" have been defined on data object or custom ones can be defined. Peach can also automatically recalculate checksums if required. Further XML blocks describe the state transitions of the system to be modelled. Peach is notable for its remote monitoring agent that examines when/if the program has crashed and can log and resume testing from that point. Its XML based code is powerful but harder to learn than SPIKE, for example.

Peach is a maturing product still under active development. The current version 2.3.8 will soon be replaced by version 3 which is a rewrite from the ground up to take advantage of the .NET platform (using mono for cross-platform support). Development has been focused on the Windows platform and currently some of the functionality does not work (or operates in a degraded way) on Linux.

Peach is being used to detect exploits in the real world but most of the work is behind closed doors. Peach has been used to find a Denial-Of-Service exploit in SolarWinds TFTP server[16] and various potential exploits in Microsoft Office and Oracle OpenOffice[17] [18].

B.7 General Purpose Fuzzer (GPF) (2007–present)

GPF[19] had the facility to capture network traffic from pcap and construct a protocol description. Users generated traffic variations based on this traffic. It was extensible. Fuzzing options ranged from pure random to protocol tokenisation. It was extended into the Evolutionary Fuzzing System (EFS) which used a Genetic Algorithm to generate the semi-valid data. It does not require source code, but statically pre-analyses the binary to identify function addresses and structures. The system used a debugger to monitor the coverage of the code within the system and the fitness function of the GA was based on this coverage metric. EFS is classed as a Grey-Box solution due to its debugger metric system.

B.8 Sulley (2008–present)

Sulley[20] adopts some ideas from the previous tools but is an evolutionary tool that aims to be easier to use. It is capable of accepting pcap and replaying it. It uses data blocks and simple primitives as per SPIKE, with checksum support for these blocks. It supports reuse of complex code objects and can model complex state machines. It uses the PaiMai/PyDBg

[15]http://peachfuzzer.com/
[16]http://www.nullthreat.net/2011/01/fuzzing-with-peach-running-fuzz-part-3.html
[17]http://www.cert.org/blogs/certcc/2011/04/office_shootout_microsoft_offi.html
[18]http:// dankaminsky.com/2011/03/11/fuzzmark/
[19]http://www.vdalabs.com/tools/efs_gpf.html
[20]http://www.fuzzing.org/wp-content/Amini-Portnoy-BHUS07.zip

process for establishing code coverage (as per EFS) and the debugger monitor and restore concept from Peach (but using a VMWare snapshot method).

B.9 Flashboom (2009)

Flashboom[21] is a simple, single function Adobe Flash fuzzing tool. It focuses on a specific undocumented function, ASNative, to operate. It is aimed at locating errors in the Adobe Flash Player. Flashboom compiles an Adobe Flash file that tests the undocumented Flash ASNative() function[22]. This undocumented function provides direct access to the internal function table within a Flash Player instance. A fuzzing server(in the form of a Ruby script, provided) calls out to the compiled flash file, calling the ASNative function with different malformed arguments. The flash file executes the function call and responds. Should a crash occur, the sent argument is retrievable and the Flash debuggers attached to the web browser can pinpoint the error. Flashboom was developed by H.D.Moore.

B.10 Spider Pig (2010–present)

Spider Pig[23] is a relatively simple single purpose Javascript fuzzer for PDF readers. The user specifies the Javascript functions for testing and Spider Pig creates PDFs containing multiple instances of this fuzzed content. The user must then independently run these fuzzed documents in a PDF reader with appropriate debugging. Spider Pig is not a framework and is not easily expandable.

[21]http://blog.metasploit.com/2009/01/fuzzing-flash-for-fun-asnative.html
[22]http://osflash.org/flashcoders/undocumented/asnative
[23]http://code.google.com/p/spiderpig-pdffuzzer

Appendix C Experimental Results

C.1 Peach

The following XML file is a simple example of a Peach pit to fuzz Microsoft Word 2003 on Windows XP. It generates a mutated .doc file on disk and launches Word to load it. Fuzzing runs continuously and is monitored by the standard Peach Agent running in a separate process.

word-peach-doc.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<Peach version="1.0" author="DSTO-DRJG" description="MS Word 2003 fuzzing definition">

  <!--
     A simple MS word document fuzzing definition.
     Mutates a sample MS word document file and passes it to MS word.
     Each word document file is named differently to avoid problems with Microsoft's
     automatic document recovery algorithms. This functionality requires changes to
     the Peach source code.
  -->

  <!-- Import defaults for Peach instance -->
  <Include ns="default" src="file:defaults.xml" />

  <!-- Define the basic structure of the html file -->
  <DataModel name="RootDataModel">
    <Block name="Block1">
      <Blob valueType="hex" value="D0 CF 11 E0 A1 B1 1A E1 00 00 00 00 00 00 00 00
         ... hex dump of existing word document to be mutated ...
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00"/>
    </Block>
  </DataModel>

  <!-- Define how to run a single test -->
  <StateModel name="State" initialState="State1">
    <State name="State1">
      <!-- Run the RootDataModel to generate the output file -->
      <Action type="output">
      <DataModel ref="RootDataModel"/>
      </Action>
      <!-- Close the output file -->
      <Action type="close"/>
      <!-- Launch the file consumer -->
      <Action type="call" method="DoTheFuzzTestNow"/>
    </State>
  </StateModel>
  <!-- Define an agent to monitor the testing -->
  <Agent name="LocalAgent">
    <Monitor class="debugger.WindowsDebugEngine">
      <Param name="CommandLine" value="C:/Program Files/MicrosoftOffice2003/OFFICE11/
        winword.exe C:/Dev/Peach-2.3.8-src/fuzz-word/input/FuzzTest-%d.doc" />
      <Param name="StartOnCall" value="DoTheFuzzTestNow" />
      <Param name="SymbolsPath" value="C:/WINDOWS/symbols" />
    </Monitor>
    <Monitor class="gui.PopupWatcher">
      <Param name="WindowNames" value="Microsoft Office Word" />
      <Param name="CloseWindows" value="true" />
      <Param name="TriggerFaults" value="false" />
    </Monitor>
    <Monitor class="process.PageHeap">
      <Param name="Executable" value="winword.exe"/>
    </Monitor>
  </Agent>

  <!-- Define the test -->
  <Test name="FuzzTest">
    <Agent ref="LocalAgent"/>
    <StateModel ref="State"/>

    <!-- Define the publisher to write to file -->
    <Publisher class="file.FileWriterLauncher">
      <Param name="filename" value="C:/Dev/Peach-2.3.8-src/fuzz-word/input/
        FuzzTest-%d.doc" />
      <Param name="debugger" value="true"/>
    </Publisher>
  </Test>

  <!-- Define what to run -->
  <Run name="DefaultRun">
    <Test ref="FuzzTest" />

    <!-- Define a logger to write to disk -->
    <Logger class="logger.Filesystem">
      <Param name="path" value="C:/Dev/Peach-2.3.8-src/fuzz-word/logs" />
    </Logger>
  </Run>
</Peach>
```

To start Peach:

```
peach.bat word-peach-doc.xml
```

The Peach logger outputs a basic log file which lists the tests which detected faults. The following test was started at 9:17 and detected two faults before being forcibly terminated at 11:50 using the task manager.

status.txt:

```
Peach Fuzzer Run
=================

Command line: peach.py fuzz-word\word-peach-doc.xml
Date of run: Thu Jul 07 09:17:28 2011
SEED: 1309994983.45
Pit File: word-peach-doc.xml
Run name: DefaultRun

Thu Jul 07 09:17:30 2011:
Thu Jul 07 09:17:30 2011: Test starting: FuzzTest
Thu Jul 07 09:17:30 2011:
Thu Jul 07 09:17:30 2011: On test variation # 1
Thu Jul 07 10:10:50 2011: Fault was detected on test 255
Thu Jul 07 10:12:50 2011: Fault was detected on test 267
Thu Jul 07 10:58:40 2011: On test variation # 512
Thu Jul 07 11:50:45 2011: Fault was detected on test 773
Thu Jul 07 11:50:47 2011: FORCED EXIT OR CRASH!
Thu Jul 07 11:50:47 2011: Last test #: 773
```

The basic log file is useful as a first pass check and additional log files provide more detail on any detected faults. The test above produced the following directory structure of files:

```
Peach-2.3.8-src
 |
 +--fuzz-word
    |
    +--logs
       |
       +--word-peach-doc.xml_2011Jul06234728
          | status.txt
          |
          +--Faults
             |
             +--AgentConnectionFailed
             |  |
             |  +--773
             |    data_1_output_Named_32.txt
             |    data_2_call_Named_34.txt
          |
          +--UNKNOWN_TaintedDataControlsBranchSelection_0x19050c05_0x313f0c36
             |
             +--255
             | data_1_output_Named_32.txt
             | data_2_call_Named_34.txt
             | LocalAgent_StackTrace.txt
             +--267
               data_1_output_Named_32.txt
               data_2_call_Named_34.txt
               LocalAgent_StackTrace.txt
```

The results of each Peach run are logged in a separate timestamped directory containing the “status.txt” file. If any faults are detected a directory is created for each unique fault, and within this another directory for each test that caused that specific fault along with input files and stack traces. This is helpful because you can easily compare different inputs which resulted in the same fault.

The following “LocalAgent_StackTrace.txt” file explains the detected fault from test 267 in more detail. The following output from Microsofts ”!exploitable” tool suggests that this a potentially exploitable fault as data from the faulting address is later used to determine whether or not a branch is taken.

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:/Program Files/MicrosoftOffice2003/OFFICE11/winword.exe C:/Dev/
  Peach-2.3.8-src/fuzz-word/input/FuzzTest-267.doc
Symbol search path is: C:/WINDOWS/symbols
Executable search path is:
ModLoad: 30000000 30bb1000   winword.EXE
ModLoad: 7c900000 7c9b2000   ntdll.dll
|.
.  0 id: 9f4 create name: winword.EXE
<Snip>

INSTRUCTION_ADDRESS:0x0000000030caf542
INVOKING_STACK_FRAME:0
DESCRIPTION:Data from Faulting Address controls Branch Selection
SHORT_DESCRIPTION:TaintedDataControlsBranchSelection
CLASSIFICATION:UNKNOWN
BUG_TITLE:Data from Faulting Address controls Branch Selection starting at mso!MsoCchWz
  Len+0x0000000000000012 (Hash=0x19050c05.0x313f0c36)
EXPLANATION:The data from the faulting address is later used to determine whether or not
  a branch is taken.!msec.exploitable -m
IDENTITY:HostMachine\HostUser
<Snip>
```

References

Aitel, D. (2002) http://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt.

Alverez, S. (2009) Commercial fuzzer and open source comparison, `http://www.whitestar.linuxbox.org/pipermail/fuzzing/2009-July/000556.html`.

Avgerinos, T., Cha, S., Hao, B. & Brumley, D. (2011) Aeg: Automatic exploit generation, NDSS.

Backus, J. (1959) The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference, *Proceedings of the International Comference on Information Processing, 1959* .

Beddoe, M. (2005) Network protocol analysis using bioinformatics algorithms.

Cadar, C., Dunbar, D. & Engler, D. (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, *in Proceedings of the 8th USENIX conference on Operating systems design and implementation*, USENIX Association, pp. 209–224.

Cui, W., Kannan, J. & Wang, H. (2007) Discoverer: Automatic protocol reverse engineering from network traces, *in Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, USENIX Association, p. 14.

Dai, H., Murphy, C. & Kaiser, G. (2010) Configuration fuzzing for software vulnerability detection, *in 2010 International Conference on Availability, Reliability and Security*, IEEE, pp. 525–530.

DeMott, J., Enbody, R. & Punch, W. (2007) Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing, *BlackHat and Defcon* .

Forrester, J. & Miller, B. (2000) 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*, USENIX Association, pp. 6–6.

Ganesh, V., Leek, T. & Rinard, M. (2009) Taint-based directed whitebox fuzzing, *in Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society, pp. 474–484.

Godefroid, P. (2010) From blackbox fuzzing to whitebox fuzzing towards verification, `http://selab.fbk.eu/issta2010/download/slides/Godefroid-Keynote-ISSTA2010.pdf`.

Godefroid, P., Kiezun, A. & Levin, M. (2008) Grammar-based whitebox fuzzing, *in ACM SIGPLAN Notices*, Vol. 43, ACM, pp. 206–215.

Godefroid, P., Levin, M., Molnar, D. et al. (2008) Automated whitebox fuzz testing, *in Proceedings of the Network and Distributed System Security Symposium*, Citeseer.

Grove, D., Murray, T., Owen, C., North, C., Jones, J., Beaumont, M. & Hopkins, B. (2007) An overview of the annex system, *in Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, IEEE, pp. 341–352.

Hanford, K. (1970) Automatic generation of test cases, *IBM Systems Journal* **9**(4), 242–257.

Howard, M. (2007) Lessons learned from the animated cursor security bug.

Kaksonen, R. (2001) *A Funtional Method for Assessing Protocol Implementation Security*, VTT, Technicial Research Centre of Finland.

Larsson, N. & Moffat, A. (1999) Offline dictionary-based compression, *in dcc*, Published by the IEEE Computer Society, p. 296.

Lehman, E. & Shelat, A. (2002) Approximation algorithms for grammar-based compression, *in Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, pp. 205–212.

Microsoft (2011) *Security Development Lifecycle: SDL Process Guidance*, Microsoft.

Miller, B., Fredriksen, L. & So, B. (1990) An empirical study of the reliability of unix utilities, *Communications of the ACM* **33**(12), 32–44.

Miller, B. P. (1988) Cs 736 project list.

Miller, B. P., Koski, D., Pheow, C., Maganty, L. V., Murthy, R., Natarajan, A. & Steidl, J. (1995) *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*, Technical report.

Muniz, S. & Ortega, A. (2011) Fuzzing and debugging cisco ios.

Myers, G. J. (2004) *The Art of Software Testing 2nd Ed*, John Wiley and Sons.

Nethercote, N. & Seward, J. (2007) Valgrind: A framework for heavyweight dynamic binary instrumentation, *in Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007*, Association for Computing Machinery.

Röning, J., Laakso, M., Takanen, A. & Kaksonen, R. (2002) Protos - systematic approach to eliminate software vulnerabilities, http://www.ee.oulu.fi/research/ouspg/PROTOS_MSR2002-protos.

Sparks, S., Embleton, S., Cunningham, R. & Zou, C. (2007) Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting, *in Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, IEEE, pp. 477–486.

Sutton, M. & Greene, A. (2005) The art of file format fuzzing, http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sutton-greene.pdf.

Takanen, A. (2009*a*) Commercial fuzzer and open source fuzzer comparison, `http://www.whitestar.linuxbox.org/pipermail/fuzzing/2009-July/000552.html`.

Takanen, A. (2009*b*) Fuzzing: the past, the present and the future, Symposium sur la securite des technologies de l'information et des communications.

Takanen, A., Demott, J. D. & Miller, C. (2008) *Fuzzing for Software Security and Quality Assurance*, Artech House.

Viide, J., Helin, A., Laakso, M., Pietikäinen, P., Seppänen, M., Halunen, K., Puuperä, R. & Röning, J. (2008) Experiences with model inference assisted fuzzing, *in Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*, USENIX Association, pp. 1–6.

Wang, T., Wei, T., Gu, G. & Zou, W. (2010) Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, *in Proceedings of the 31st IEEE Symposium on Security and Privacy*, IEEE, pp. 497–512.

Wood, D., Gibson, G. & Katz, R. (1990) Verifying a multiprocessor cache controller using random test generation, *Design & Test of Computers, IEEE* **7**(4), 13–25.

Yang, X., Chen, Y., Eide, E. & Regehr, J. (2011) Finding and understanding bugs in c compilers.

THIS PAGE IS INTENTIONALLY BLANK

Page classification: UNCLASSIFIED

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. CAVEAT/PRIVACY MARKING
2. TITLE Fuzzing: The State of the Art	3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)
4. AUTHORS Richard McNally, Ken Yiu, Duncan Grove and Damien Gerhardy	5. CORPORATE AUTHOR Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia

6a. DSTO NUMBER DSTO–TN–1043	6b. AR NUMBER 015-148	6c. TYPE OF REPORT Technical Note	7. DOCUMENT DATE February, 2012

8. FILE NUMBER 2011/1216182/1	9. TASK NUMBER 07/343	10. TASK SPONSOR DSTO	11. No. OF PAGES 43	12. No. OF REFS 37

13. URL OF ELECTRONIC VERSION `http://www.dsto.defence.gov.au/publications/scientific.php`	14. RELEASE AUTHORITY Chief, Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

Fuzzing
Automated Vulnerability Discovery
Computer Security

19. ABSTRACT

Fuzzing is an approach to software testing where the system being tested is bombarded with test cases generated by another program. The system is then monitored for any flaws exposed by the processing of this input. While the fundamental principles of fuzzing have not changed since the term was first coined, the complexity of the mechanisms used to drive the fuzzing process have undergone significant evolutionary advances. This paper is a survey of the history of fuzzing, which attempts to identify significant features of fuzzers and recent advances in their development, in order to discern the current state of the art in fuzzing technologies, and to extrapolate them into the future.

Page classification: UNCLASSIFIED