On Designing an Efficient Distributed Black-Box Fuzzing System for Mobile Devices

Wang Hao Lee
Institute for Infocomm Research
Singapore
whlee@i2r.a-star.edu.sg

Murali Srirangam Ramanujam
Institute for Infocomm Research
Singapore
muralism@i2r.a-star.edu.sg

S. P. T. Krishnan
Institute for Infocomm Research
Singapore
krishnan@i2r.a-star.edu.sg

ABSTRACT

Security researchers who jailbreak iOS devices have usually crowdsourced for system level vulnerabilities [1] for iOS. However, their success has depended on whether a particular device owner encountered a crash in system-level code. To conduct voluntary security testing, black-box fuzzing is one of the ideal low-cost and simple techniques to find system level vulnerabilities for the less technical crowd. However, it is not the most effective method due to the large fuzzing space. At the same time, when fuzzing mobile devices such as today's smartphones, it is extremely time consuming to instrument mobile devices of varying versions of system software across the world. This paper, describes Mobile Vulnerability Discovery Pipeline (MVDP), a semi-automated, vulnerability discovery pipeline for mobile devices. MVDP is a carefully crafted process targeted to produce malicious output that is very likely to crash the target leading to vulnerability discovery. MVDP employs a few novel black-box fuzzing techniques such as distributed fuzzing, parameter selection, mutation position optimisation and selection of good seed files. To date, MVDP has discovered around 1900 unique crashing inputs and helped to identify 7 unique vulnerabilities across various Android and iOS phone models.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software—*Mobile Vulnerability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed Debugging*

General Terms

Design, Experimentation, Security

Keywords

Black-Box Fuzzing, Zero-Day Vulnerability, Crash Analysis, Smartphones

.

1. INTRODUCTION

The increased sophistication of mobile devices brings with it many unknown vulnerabilities. It is critical to identify the vulnerabilities before attackers use them for 0-day exploits. The simplicity of black-box fuzzers allows non-security professionals to run security experiments and gather security related defects for the manufacturer or security researchers. On the other hand, the space for blind fuzzing is intractable and many duplicated fuzzed files or files with less meaningful mutations are produced. Hence, it is vital to devise techniques for discovering unknown vulnerabilities from a black-box perspective that yield higher detection-rates with fewer input tests. At a high level, three approaches to discovering software vulnerabilities exist: white-box, black-box and grey-box testing. The most suitable approach depends on the availability of the source codes, design specifications, size of source base and time and resources available. White box testing makes use of a program's source code to test the internal mechanism of the system or component. The openess of today's mobile Operating Systems today are mixed. Apple's iOS is generally closed source with some open-sourced components like XNU, `WebKit`, Objective-C runtime, `dyld` and so on. Debugging system level binaries are difficult without a Jailbreak which in itself is an exploit. Most of Google's Android OS is fully open-source with the exception of device drivers and UI customization [2]. This means bug in the Android core system might not affect customised OEM devices and vice-versa. The less popular Blackberry 10 and Windows Phone operating systems on the other hand are completely closed source. The complexity and size of the code-base would require significant time and resources for investigation. These code bases would also get updated before a researcher can sufficiently understand the system. In the context of voluntary security testing, it would be too intrusive to instrument a volunteer's device. Even though robust desktop environments may accurately simulate the environment libraries of mobile platforms, actual live devices provide more accurate exploitation context. Furthermore, not every mobile device platform has common libraries with its desktop counterpart - for example, Webkit on iOS vs Mac OS X. This means that instrumenting a library on a desktop may not provide the same result as on a mobile device.

On the other hand, grey and black-box testing approaches require little or no knowledge of the architecture, respectively. As such, a large volume of test cases are needed to probe an unknown target. Black-box fuzzing at the extreme

end of the spectrum, requires numerous unexpected or malformed inputs to test the response of the target. Although the basic idea may seem to lack elegance, it is highly effective in practice and simple in discovering critical bugs [3] even for users not coming from a security background. An average user has nearly 3 mobile devices according to a Sophos survey [4]. Shops also have unsold display-set devices switched on during the day. To harness the availability of idle devices and allow mainstream users to help researchers find security defects, we describe in this paper techniques such as distributed fuzzing, parameter selection, mutation position optimisation and selection of good seed files.

Audio, video and images are the most consumed media types on smartphones and drive-by-download exploits have been developed for these media in the past [5, 6, 7]. Hence, for this paper, we targeted the multimedia libraries (audio, video and images) on Android and iOS. Although only audio, video and images are tested in this paper, the technique is general and can be extended to include other data-formats that can be sent over the internet.

1.1 Fuzzing Challenges and Motivations

Black-box fuzzing is a popular, cheap and effective for finding bugs in applications for the masses. However, a good scalable black-box fuzzing architecture can always present challenges that need to be addressed before it can effectively enable vulnerability discovery. Overall, we summarize several challenges as below:

a) **Selecting good starting (seed) files**: The quality of the fuzzing output depends on the quality of seed files. This means the seed files from which the malformed inputs are generated must be of good quality [8]. The GIF file format, for example has several well-defined sections defined in its specification [9]. A good seed file should have all sections and sub-types of a particular data-format. More information about this is provided in Subsection 2.1.

b) **Virtually unlimited fuzzing space**: The fuzzing space for any file format can be exponentially large and exploring every single possibility can be next to impossible. For example, fuzz testing executed using binary files has is that it requires a large number of fault-inserted files to cover every test case [10]. As such, without much knowledge of the input format, we still need a method of generating and selecting fuzzing configurations that output fuzzed files containing a uniformly distributed set of mutation positions covering a diverse set of fields of the seed file.

c) **Fuzzing scalability**: If we were to generate test cases on the device itself, it is likely that the same malformed file is generated across multiple devices. This contributes to wasted computing power and less coverage of fuzzing space. There is also no means to compare test results across devices as each device might have performed the same or a different test. Hence, the fuzzed files should be generated only once and have copies distributed to different devices for testing. We need a server setup to coordinate the fuzz distribution across a great number of devices and collect and quantify the results.

When multiple devices of the same model and OS version are being fuzzed, there is a need to devise a method of balancing the inputs. With load balancing, more tests can be processed on a device/OS platform, duplicate tests can be eliminated and vulnerabilities can be discovered faster. This means that if we have multiple devices running the same OS version, the server must be able to identify them as of the same model and and allocate different inputs to each one of them.

When fuzzing many devices across the world, it is essential to have a new set of test inputs ready for the next arriving device of the same model and OS version. Storing fuzzed input thus requires large amounts of storage space and the growth of space utilisation increases with the amount of devices connected to the test server. In addition, conducting tests across the internet requires massive network transfer of these test data - a problem when it comes to video files especially. Volunteers donating their free phone CPU cycles might think twice if their mobile network utilisation is high. Hence, we will need a means of optimizing the content being stored without comprimising on performance considerably.

With these challenges, it is noted that more efficient black-box fuzzing framework should be developed for discovering unknown vulnerabilities and bugs in mobile platforms, especially on smartphones.

1.2 Contributions

In this paper, we address the challenges described above and designed *Mobile Vulnerability Discovery Pipeline (MVDP)*, a scalable distributed fuzzing infrastructure for discovering unknown vulnerabilities for smartphones.MVDP comprises of techniques to enable internet-scale voluntary security testing. The paper emphasizes the following key contributions for improvising black-box testing:

- **Ensuring universal uniqueness of fuzzes.** Sending the same file to the same device is a wastage of time, computing and network resources. Our paper presents FEET (Fuzzing Engine Evaluation Tool), a novel mechanism that ensures unique fuzzed files with well-distributed modification positions are prioritised.
- **Ensuring uniformity of fuzz campaigns (jobs).** Fuzzing is phased as jobs. Jobs are groups of `N` files analagous to a campaign in [11]. The basic idea of an optimized job in our case is that a job will contain a high diversity of modification positions as possible. Without considering too many intricacies of the file-format or system code from a black-box perspective, this technique attempts to ensure that a job tests out as many "fields" of a data-format as possible. This uniformity is measured using a chi-square test on all files in a job/fuzz epoch.
- **Improving Scalability and Fuzz Reuse.** It is mentioned in Section 1.1 that fuzz files are generated once and the copy is used to fuzz many devices, the side effect of it is the staggering growth of storage use when devices run out of test cases or arrival of more test devices. Diff-patches rather than use of actual file has

a huge impact especially when it comes to MP4 and other storage-heavy files. Before fuzzing, the device creates the actual fuzz file by patching out the seed file. Refer to subsection 3.1.1 for more information on this approach.

By sending just the patches of the seed file rather than sending fuzzed files, we have been able to achieve up to 90% lesser network traffic. Each fuzzed file is only a BSDdiff patch file of approximately 500bytes. This makes it suitable for large scale fuzzing across the internet and future voluntary experiments by the general public on a large scale analagous to SETI@home [13] or BOINC [12].

Overall, MVDP comprises of various stages to generate good fuzzing output, hasten the process of vulnerability discovery and maximize the coverage of the test space. MVDP at its present stage discovers vulnerabilities of media file-formats but can be applied to target any data-format acessible over the internet.

The layout of the paper is organised as follows. Section 2 introduces the architecture of MVDP and its four operational stages. In Section 3 we share the implementation and experiments done. Section 3 sheds light on the results obtained and its implications. Section 4 lists related work, Section 5 talks about future work and Section 6 concludes the paper.

2. THE PROCESS

The Mobile Vulnerability Discovery Pipeline consists of several processes that can be programmed into server-end and device-end. Figure 1 provides an overview of the pipeline with the numbers representing the order of operations. The entire MVDP process can be described as *three* operating stages: the Quality Input Generation, Device and Task Management, and Crash Analysis.

2.1 Quality Input Generation

This is a semi-automated stage consisting of automatic downloading and initial good seed file selection. Additional manual augmentation of good seed files is advisable but optional. It aims to construct the best fuzzing configuration to be used such that the fuzzed files have good coverage and uniform distribution. The following sub-sections describe them in more detail.

2.1.1 Seed File Downloader (SFD)

To fuzz a data-format effectively, the clean input data (also known as seed files or initial files [14]) must cover as high field coverage as possible as defined in the respective file format RFC. To achieve this, data-format samples are downloaded and further manually augmented.

For the purpose of this paper, the main data inputs for fuzzing are file-based inputs. Rather than constructing such a file from scratch, files are downloaded from the internet. The search and download process is automated using web APIs. We scan the Internet for seed files using the Google custom search Engine [15] and Microsoft Azure Search [16] to obtain raw inputs which are then given to SFAT.

2.1.2 Seed File Analysis Tool (SFAT)

SFAT compares, sorts and clusters the raw input files according to their protocol coverage. File format parsers like

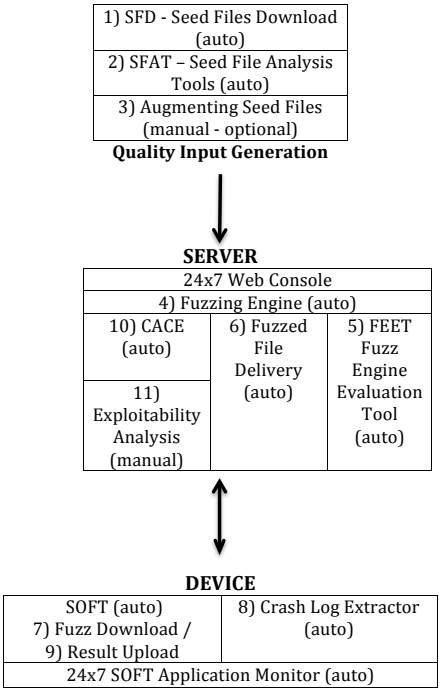

Figure 1: Architectural Flow Diagram of MVDP showing automatic and manual processes.

Hachoir [17] provides file information for the seed file selection algorithm (more in section 2.1.3) to calculate file format specification (RFC) coverage for a particular file. The files containing high data-format coverage are known as good seed files. In some protocols, there are mutually exclusive fields; each forming a sub-type of a file format. For example, in the case of PNG files, there are RGB and ICCP PNG file sub-types [18]. A PNG file can contain only either one of the ICCP or the sRGB chunk. In this case, SFAT automatically ignores files with field-coverage being total subsets of another downloaded file so sub-types are covered.

2.1.3 Seed File Selection Algorithm

SFAT is the tool to be run when a new binary file format is to be fuzzed using mutation fuzzing. It examines all seed files by using the following heuristics in the following order:

1. Top-Level Domain Score (TLD)
2. Field Score (F)
3. Field Occurrence Score (D)
4. Occurrence Distribution Score
5. File Size Score

The scoring is done for each file format separately and heuristics are used hierarchically. For the file formats we tested, there are (usually) many seed files with equally high TLD score; for example, internet downloaded PNG seed files with highest TLD scores have all 3-4 critical chunks [18] and 6 more ancillary chunks. These files with highest TLD scores are further ranked using Field Score. The best from Field Score is ranked by Field Occurrence Score and so on. If more than one seed file reaches the 5th level heuristic, the file with the smallest file size is selected.

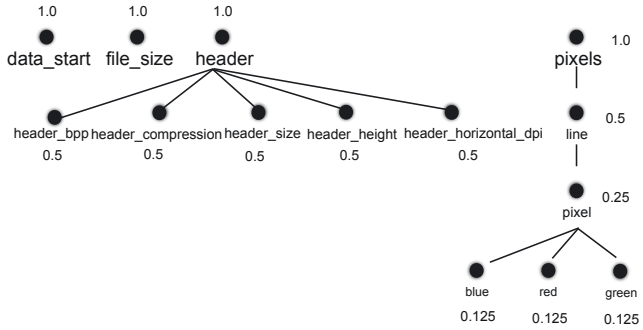

Figure 2: Field Score

(i) **Top-Level-Domain Score**

File-formats typically use markers/chunks to determine a major field. For example, PNG has 21 types of chunks out of which only 3 to 4 are mandatory [18]. Typically, the higher the number of top-level markers/chunks are, the better the seed file candidate.

(ii) **Field Score**

Given an instance of a file type F where there are n top-level domain fields we define the fields of file F as $< f_i.....f_n >$.

A weight is assigned to a field/sub-field to denote the significance of a field in a file-type instance. This weight value assignment scheme is experimental and can be changed accordingly. Within f_i where $i < n$, the set of weights for each subfields of a field f_i is denoted by $S_i =< s_{ij}...s_{im} >$, where m is the number of subfields of the field f_i.

The weight for each subfield is defined as s_{ij}. It is defined as: $s_{ij} = \frac{1}{(2^{l-1})}$ where l is the level of the field in the subfield tree.

By definition, top-level domain fields always have a weight of 1. The weight value decreases with tree height. For example, in BMP file format, the weight of a field called used_colors, (header/used_colors) is a second level sub-field and 'header' is the top-level-domain field. $\frac{1}{2^1} = 0.5$

Consider another example, if its a 4 level field e.g. (/pixels/line/pixel/red) $\frac{1}{2^3} = 0.125$

The total weight of a field f_i, denoted w_i is defined as:

$$w_i = \sum_{j=0}^{m} s_{ij} \tag{1}$$

Given the definitions above, we define the Field Score of file F as:

$$\prod_{i=0}^{N} w_i \tag{2}$$

where w_i = a weighted sum of the sub-fields and root field defined in Equation 1.

Consider the example of a Field score of a hypothetical file in Figure 2.

$$F = dataStart * FileSize * Header * Pixels$$
$$= 1.0 * 1.0 * 3.5 * 2.125 = \mathbf{7.4375}$$

In general, a higher field score represents a better seed file.

(iii) **Field Occurrence Score**

The field occurrence shows the number of times a field/subfield appears in the file. Let the field occurrence of a sub-field s_{ij} be o_j. The subfield weighted occurrence score B is given by:

$$B_{i,j} = o_{i,j} * s_{i,j} \tag{3}$$

From Equation 3 the field weighted occurrence score C is calculated as:

$$C_i = \sum_{j=0}^{m} B_j \tag{4}$$

Where m is the number of fields in the i^{th} field. Given B_j and C_i, we can calculate the $fieldOccurrenceScore$ D as follows:

$$D = \sum_{i=0}^{n} C_i \tag{5}$$

A higher field occurrence score corresponds to a generally better seed file as it generally denotes a better overall spec coverage given the lack of time to read RFCs.

An example of a field occurrence score calculation is shown in Figure 3.

(iv) **Occurrence Distribution Score**

In occurrence distribution score, all fields or subfields are flattened and not weighted. We define the occurrence of a field/subfield as o_k. Let T be the total number of all subfields and fields of the input file and the set of all subfields and fields be k. We define the mean field/subfield occurrence as

$$\mu = \sum_{k=0}^{T} \frac{o_k}{(T)} \tag{6}$$

The file-wide standard deviation of the occurrences is given by:

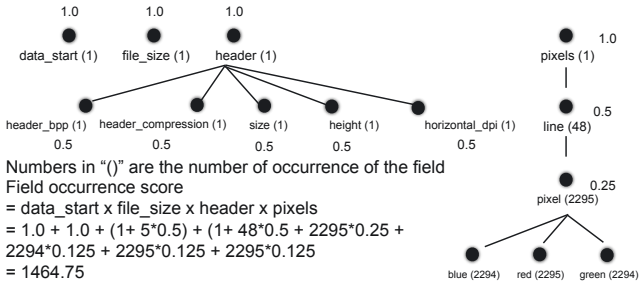

Figure 3: Field Occurrence Score

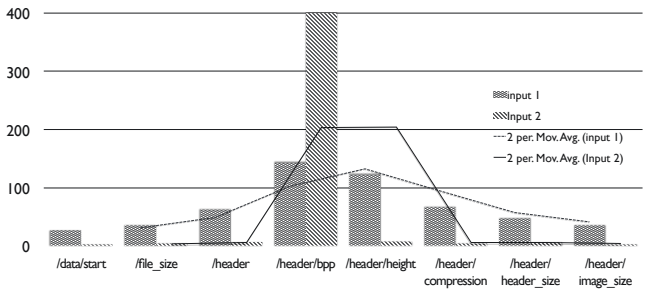

Figure 4: Occurrence Distribution Score

$$\sigma = \sqrt{(\frac{1}{n})} * (\sum_{k=0}^{T} (o_k - \mu)^2) \quad (7)$$

From Equation 6 and 7, the occurrence distribution score is calculated as:

$$occurrenceDistributionScore = \frac{\mu}{\sigma} \quad (8)$$

A higher field-wide occurence distribution score is less desirable. This is because mutation fuzzing will tend to fuzz only data in one field type. So the mean occurrence is taken into consideration. A higher occurrence distribution score is a better seed file. Figure 4 shows an example of a more desirable seed file (input 1) in terms of occurrence distribution score.

(v) **File Size Score**
Finally, after applying the above heuristics, if more than one seed file reaches this level, the best seed file is the file with the smallest size. This is because the mutations by a fuzzer will be more likely to hit a wider range of fields. FEET also works more efficiently with a seed file of a smaller size. Patching a smaller seed file (see section 2.2.1) to produce a fuzzed file will also be faster.

The selected seed files are still downloaded from the internet and may not contain rare fields that are less commonly used. The combination of SFD and SFAT seeks to reduce the amount of rarer fields to be introduced by automatically selecting a high coverage seed file as a starting point. One can then instrument execution of a open source library supporting the data format (for example `libSkia` [19] is to PNG) to get maximum coverage. However, this technique is platform and application library dependant. Good code coverage in Skia on Android might not directly imply good code coverage on `ImageIO` [20] for iOS.

2.1.4 Fuzzing Engine

Seed files selected by SFAT are fed into a fuzzing engine and the results are evaluated to determine the best fuzzing configuration to use with test devices. Since the fuzzing is done on server end, another fuzzing engine can be swapped out without affecting the SUT (system under test) in a remote corner of the internet. The fuzzer we created for this set of experiments is called Fe2 that mutates files with operators that mutate meta-data targeted at parsing logic.

Fe2 makes use of FEET described in Section 2.1.5 to automatically ensure that seed files have their mutations spread out throughout the protocol. Spreading the mutations throughout the protocol ensures that the fuzzer tests the handling of the file in all its protocol regions which may improve vulnerability detection due to the potential increase in code coverage.

Several different file fuzzing operators were implemented in Fe2. The operators are file-format agnostic so they can be used on several file formats.

(a) **Remove random string** - Removes a random section of random size and location in the input file. The objective is to stress the file parser by removing essential and nonessential parts of the input.

(b) **Add random string** - Adds a random string of random size to a random location. The objective is to add unexpected sections to the input file.

(c) **Change random string** - Changes multiple random sections. Randomizes the number and size of the sections, and the contents to be substituted. The objective is to determine if the file parser could recover from multiple errors in the input file.

(d) **Change random characters** - Replaces characters in the input file with random characters. Randomizes the number of locations selected for replacement and the replacement characters. The objective is to introduce unexpected characters.

(e) **Change cases** - These 2 operators seeks out and invert either the lowercase or uppercase ascii characters in the input file. It replaces single characters at one or more locations. Replaces one or more number of characters.

(f) **Replace null characters** - Finds null characters and replaces them with character A. This is similar to mutation operation (g) with the difference being single null characters instead of double. This operation probes libraries written in traditional C-style dynamic memory allocations with null-terminated strings. It aims to trigger buffer overflow errors in the parser.

(g) **Replace null string** - Finds instances of double null characters and replaces them with AA. The objective is to test file parsers dependent on double-null delimiters in C++ coded programs.

2.1.5 Fuzzing Engine Evaluation Tool

In order to determine the superiority of the fuzzed output, we have devised a Fuzzing Engine Evaluation Tool (FEET). FEET considers fuzzing parameters, uniqueness of the job and uniformity of the fuzzing space covered to evaluate the quality of the fuzzed output. Here we define a job to be a set of fuzzed files which were fuzzed with a particular fuzzing configuration. This notion is similar to a 'campaign' in [21] except larger numbers of up to 10000 files can be used in 1 job.

We define three levels of uniqueness for grouping of fuzzed input. They are local, global and universal uniqueness. The best fuzzing configuration has most files with universal uniqueness. All files for one file-type in MVDP have been selected to be globally unique.

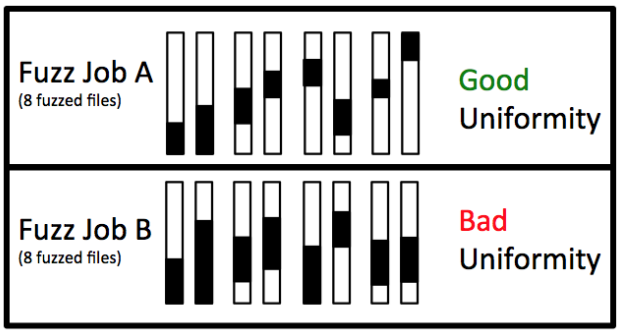

Figure 5: Different modification position distributions between 2 fuzz jobs (Black areas in a file are mutations)

a) **Local Uniqueness**

Local uniqueness is the percentage of unique fuzzed files in a job. Uniqueness of a file is determined by comparing the hash of the file against the hashes of all the remaining files within a job.

b) **Global Uniqueness**

There are usually millions of fuzzed files generated and packaged in several hundred jobs. Let D and G be 2 jobs with all locally unique files. Let d_i be a file in a job D and g_j be a file in job G, where $i, j > 0$. It is possible that $d_i \equiv g_j$ or even $D \equiv G$. Hence, for a fuzzed file to be globally unique, it must be unique across all jobs produced from a given fuzzer regardless of fuzzing parameters.

c) **Universal Uniqueness**

As MVDP is able to utilize several different fuzzers to produce fuzzed output, it is then possible for two fuzzers to produce 2 identical files. A fuzzed file is considered universally unique if it is both locally and globally unique and an identical copy of the fuzzed file is also not produced by another fuzzer within MVDP.

d) **Uniformity**

Fuzzing needs to concentrate on a specific location of the targeted data-type within a job. If the fuzzing is wildly distributed throughout the file, critical sections of files might be missed. Hence, we deduced that in each job, the fuzzing needs to be concentrated yet uniformly spread out in one particular section. As such, the definition of uniformity here refers to having a 'goodness-to-fit' between distributions of modified positions in a fuzz job against that of an even distribution of positions. An example of an evenly distributed fuzzed job and a non-evenly distributed fuzzed job is as shown in Figure 5.

The uniformity for seed file modification locations is measured using the chi-square test. We used the chi-square statistic χ^2 for a fuzzed job with k modifications as the uniformity distance:

$$\chi^2 = \sum_{1 \le i \le k} \frac{(Y_i - E)^2}{E} \tag{9}$$

where $Y_1, Y_2, \ldots, Y_k$ are the modifications positions in the fuzzed file and the expected position of each modification is E (where $1 < i < k$). The expected value E is given as:

$$E = \frac{\sum_{i=1}^{k} Y_i}{K} \tag{10}$$

After obtaining the chi-square distance for each job, we calculated the chi-squared probability using the uniformity distance with the degree of freedom being $K-1$. We ensured that all fuzzing jobs have a chi-squared probability > 0.05 to be considered uniform.

For each file format, we let FEET generate and analyse multiple jobs with various fuzzing configurations and select fuzzed jobs ranked by their chi-squared probability and uniqueness.

2.2 Device and Task Management

To test the Mobile OS's resistance to the malformed files, we would need to open/render each file individually and log the results. With the vast quantity of fuzzed files to be tested, we needed an automated system to deliver jobs to the devices, collect and consolidate results. Hence, we built the Security Testing Arsenal for Mobile Platforms (STAMP).

2.2.1 STAMP

STAMP is a server based system that automatically coordinates the flow of generating fuzzing jobs, distributing them to the devices, load balancing the work among devices of the same type and collating the results for *centralised* graphical analysis and evaluation.

The problem of duplicate test cases is solved by generating the jobs within STAMP and distributing them to every new device to be tested. This also removes the waiting time for fuzzed files to be generated when a new device registers with STAMP. When a device registers with STAMP, there are already a large number of fuzzed inputs downloadable so the device can be tested without delay.

STAMP server holds history of all fuzzed files for newly registered unseen devices and generates new fuzzed file for seen devices. On a large scale of mobile clients, requests for fuzz files arrive faster than fuzz generation. The server then has to churn fuzz output at full-capacity creating exponential storage space growth. In addition, mobile network transfers are slow and less reliable especially for bigger file formats. To delay the rate of increase of storage use and reduce the amount of network data usage, we made use of the binary diff/patch utility. We generated patches between the fuzzed files and the Super Seed Files and stored the patches instead. This reduced the used space by more than 90%. This also greatly reduced the amount of network traffic (when sending the file from server to devices) and traded the download time for mobile device CPU's patching time. Today's mobile devices have sufficiently capable processors for patching files. Hence, this trade-off is worthwhile.

2.2.2 STAMP clients

In the MVDP architecture in Figure 1, we mentioned about SOFT clients. These are small applications called SOFT (STAMP on-device Fuzzing Tool). SOFT clients are developed for each mobile platform. The client is easily portable to other mobile platforms if needed. The SOFT clients automatically interacts with the STAMP server to download jobs and patch them (since the fuzzed files in the

server are stored as diff patches). Once a job is downloaded and patched, the SOFT client renders the fuzzed input using the native libraries/APIs provided by the OS. Upon completion of a job the clients reply to the server with the job results.

2.3 Crash Analysis

STAMP has a web interface which the security researcher can visit to obtain information about the fuzzing progress. The researcher can tell exactly which file caused a crash in a particular device at a particular time. All the STAMP reported crashes can be reproduced manually using the default application provided by the respective mobile OS. When doing so, logs on the mobile device (for example - Android) are monitored for any crashes. For example, image files are opened using the default Gallery application in Android. The Android system logs kernel errors during a crash into a Tombstone report; which are automatically extracted by using a shell script for further analysis. Similarly with iOS, the logs registering the stack trace, register values and fault locations during a crash are captured.

Inspection of crash dumps - Manual binning of all crash dumps is not scalable and is time consuming. With the Crash Automatic Classification Engine (CACE), this process has been automated. CACE processes the reports from Android's kernel crash logs (also known as tombstone) and iOS's crash logs to generate bins of unique vulnerabilities based on the type of crash. The bins are - Exploitable, Non-exploitable, Potentially Exploitable, Potentially non-exploitable or DOS. After CACE bins the crashes, researchers can focus on exploitable or potentially exploitable cases.

The key features of the engine are as follows:

- CACE is a rule based system to identify unique vulnerabilities that could have been exposed by multiple malformed files. The crash dumps are examined for terminated signal, fault address, terminated code (if exists) and the call stack along with the program counters. Hash value of all these entities combined form a unique ID for the vulnerability.

- CACE identifies the source of crash in binary (for iOS and Android) and in source locations (for Android). For example, if a crash occurs when opening an mp4 file on Android, CACE is able to point to the location in the source code/binary that led to the crash from the stack trace.

The unique vulnerabilities are then manually inspected for security issues. With the help of a scriptable disassembler. We inspect the libraries at the fault location to determine the nature of crash and severity level.

Depending on the type of access, crashes may or may not be exploitable. Various exception types and exploitability levels exist and we explored the approaches of CrashWrangler and Microsoft's !exploitable [22] to derive the common set of conditions for our own implementation; especially for Android where no crash triaging tool is available for security related purposes. In general, the SIGSEGV (11), SIGBUS (10) and SIGILL (4) on Unix flavoured Operating Systems such as Android and iOS are considered interesting to our experiments as they mean invalid memory access at the user space or kernel space and are possibly an indication of execution of data segments. Thus CACE automatically first filters out crashes containing these signals for exploration.

Next, based on the crashing address we found at the point of crash, we determine the instruction type last executed before the crash. Several types of ARM instruction classes correspond to read, write or execute instructions. Some examples are given below:

```
ldr r0, r1, r2;  - read
bl r1;           - execute
mov r1, [r4];    - write
```

When the last executed instruction before crashing is an unprivileged write or execute instruction, it is considered as potentially exploitable. Comparatively, crashes where the last instruction is an unprivileged read are not-likely exploitable.

2.4 Vulnerability Analysis

The degree of control the fuzz input has on exploitability is examined manually after a crash is binned by CACE. For every fuzzed input that corresponds to a potentially exploitable crash, we search for another existing crash with the same stack trace. For these two crashes with the same stack trace, we manually compare the crashing address value and its address contents/register value to be written or executed. If these address values or contents are relatively similar (ASLR slides library start addresses), then fuzz input does not influence the crash to a sufficently large extent and is henceforth considered as less interesting. On the other hand if the address/values for the similar crashes are all different, it means the crash is at least partially dependent on the given fuzzed input.

From time to time, it might not be possible to identify at least 2 potentially exploitable crashes with the same stack trace. When this happens, a diff operation is performed on the crashing input against the original seed file. The differences are recorded as we slowly move from the crashing input to the original seed file removing irrelevant file modifications. We stop removing modifications when we reach an input with a minimal set of modifications that produces the same crashing stack trace. When moving from the original crashing file to the minimal modification crashing file, the crashing address value and address contents/register values are examined for each crash. If the crash addresses or values are different during the minimisation process, this crash can be controlled from a given fuzzed input and is considered to be a crash where an exploit can be made.

3. IMPLEMENTATION AND EVALUATION

In this section we describe the experiment setup and discuss some of the important results obtained from MVDP.

3.1 Experimental Setup

All of our experiments were run on STAMP which is implemented with a server running Ubuntu 12.04 on an Intel Xeon E5-2697 v2 @ 2.70GHz 64 GB RAM. SFD, SFAT, Fe2 and FEET are implemented as Python scripts with the Python NumPy library. We modified hachoir-urwid [17] to provide SFAT with the structure of a recogonised file format in a hierarchical form. STAMP has been developed using Python's Django Framework with an interactive web-based

	Without BSDiff	With BSDiff
Storage Space	9.8mb	60kb
Fuzzing Time (if no crash)	1018s	1105s
Creation Time	63s	1864s
Network transmission	8.38s	0.46s

Table 1: Performance hit when using BSDiff to reduce storage requirement

dashboard, connecting to a MySQL and monogb database backend. The console allows for fuzz job monitoring, addition of jobs with user-preferred configuration, assignment of jobs to the devices, test device management, device progress monitoring, user administration and statistical visualization of crashes. HiCharts library was used to give a graphical representation of the number of crashes discovered and rate of discovery. This aids the security researcher in modifying the fuzzing configurations to better target vulnerable formats and regions.

The SOFT clients are developed using the Android and iOS SDKs. They are running on real devices or emulators. 5 workstations with Intel Xeon E5-2650 v2 @ 2.60GHz 32GB RAM run a total of 100 Android Emulators. We set up a fuzz farm a dozen iOS and Android devices in addition to running the Android emulators. The clients also verify any crashes discovered by the system and restart themselves upon a crash. On Android, for example, there is a service that constantly monitors the SOFT application to detect crashes. When the application is found to have crashed, the service takes the necessary steps to clean up and relaunch the application. The application then proceeds to retest the same file which caused the crash before moving onto the next test file. Information about whether the crash was reproducible is sent to the server. This has largely helped in reducing the waiting time between a crash and human intervention to restart the SOFT application. It has thus made it possible for SOFT to be running continuously for months with little/no supervision. We conducted the fuzzing experiments on multiple Android Emulators running Android 2.3.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x and 5.0 and popular devices from Samsung, LG, Motorola, Huawei and Google and all iOS devices. A total of approximately **5 million test cases** were evaluated in the system for formats PNG, GIF, JPEG, TIF, MP3 and MP4. Note that not all the devices finished processing all the jobs at the point of this paper as devices run at different speeds due to their hardware and software capabilities.

3.1.1 Storage Space Management

To mitigate the exponential disk space explosion from generating a few million fuzzed inputs, we developed a new way of producing and delivering the fuzzed input. Rather than directly generating the fuzzed input, BSDiff [23] patches are generated for each fuzzed input "diffed" against the seed file. These diff patches are stored on the server disk. Prior to fuzzing, the mobile device clients download a copy of the seed file and the BSDiff patches. The client then reconstructs the fuzzed files by applying the patch to the seed file. Finally the clients begins fuzzing by opening the files reconstructed from the patches.

3.1.2 Crash Classification

With FEET	Without FEET
95.24	86.49
95.66	81.18
99.13	83.23
98.89	87.93
97.83	79.66
96.09	79.49
95.40	84.09
96.97	85.72
96.13	84.61
95.29	98.68

Table 2: Percentage uniqueness of first 10 jobs distributed to clients

To identify the module and function that caused the crash, the stack trace - obtained by the method described previously - is parsed to get the filename of the shared library and its offset to the function. IDA pro is then invoked via a IDAPython script to read the shared library and identify the function and instruction causesing the crash. For Android devices, ADB's `logcat` output is parsed to look for crashes and CACE examines basic exploitability information. For iOS devices, CrashWrangler [24] is modified with access to ARM shared libraries and an ARM version of GDB/LLDB to triage iOS crashes.

3.1.3 Crash Similarity

For all reproducible identified crashes, a core-dump is generated and symbolicated. With the presence of ASLR, stack trace similarity is identified based on function-names call sequence in the stack backtrace along with the ARM CPSR value before crash.

3.2 Experimental Parameters

We set up a fuzz farm consisting of a dozen iOS and Android devices in addition to running more than 100 Android emulators in Ubuntu desktops. The file formats tested are GIF, JPEG, PNG, MP3 and MP4 video. The fuzzing experiments were run for 2 weeks and the crashes are analysed by CACE. As we are still investigating several crashes to discover the vulnerabilities and to prevent attackers from exploiting them, we have anonymised the device identities.

For every stage of the vulnerability discovery pipeline, we define groups of metrics for our experimental findings categorised below:

3.2.1 SFAT Metrics

We evaluated all fuzzed results across all devices based on fuzzes created from 2 different mpeg-4 seed files across all devices. One mp4 file has lower field_scores, field_occurance_scores and occurancedistributionscores compared to the second mp4 file. The number of unique crashes and total number of crashes identified for each seed file to evaluate the effectiveness of SFAT.

3.2.2 FEET Metrics

We ran some GIF fuzzing experiments without FEET on several devices running Android 4.0.4 and compared the results with FEET to ascertain its effectiveness. A total of 9 jobs with 10000 files each were tested. The job IDs are fuzzed in order - i.e., ID 1 is fuzzed first and 9 last. We derived the following metrics to measure the effectiveness:

FEET	Job ID	#Crashes	#Unique Bugs
Without FEET	1	1	1
	2	0	0
	3	0	0
	4	0	0
	5	0	0
	6	0	0
	7	0	0
	8	0	0
	9	0	0
With FEET	1	0	0
	2	0	0
	3	3	1
	4	3	1
	5	6	1
	6	0	0
	7	0	0
	8	0	0
	9	2	1

Table 3: Fuzzing on Android 4.0.4 Device 1 with and without FEET

(a) **Uniqueness**

FEET tests the uniqueness of fuzz files produced by each fuzz configuration.A job in this experiment consisted of 10,000 fuzzed files. The more uniqueness a job has, the more diverse the individual test files in it. We of course prefer to maximise the uniqueness percentage as this means we are fuzzing more variety of files which in turn promises more chances of striking a crash.

(b) **Crash Arrival Rate**

We first compare the *crash arrival rate*, i.e. how fast can a crash be obtained using FEET selected fuzzing configurations against randomly selected fuzzing configurations.

(c) **Number of Crashes**

We also compare the *number of crashes* obtained using FEET selected fuzzing configurations against randomly selected fuzzing configurations.

(d) **Variety of bugs**

How many unique bugs are discovered using FEET selected fuzzing configurations against randomly selected fuzzing configurations.

3.2.3 STAMP Metrics

We analyse the speedup using distributed fuzzing 10,000 files against a fixed number of fuzzes.

3.3 Experimental Results

Please note that we have anonymised the device identities because we are still investigating several crashes to discover vulnerabilities.

3.3.1 FEET Results

(a) **Uniqueness** Table 2 shows the percentage uniqueness of jobs distributed to clients. From Table 2, we see that out of the 100,000 files in this experiment (10 jobs * 10,000 files per job), using FEET provides an average uniqueness factor of 96.67% compared to an average uniqueness factor of 85.11% without FEET.

For illustration purposes, let us consider that testing one JPG image on an iPhone takes 1 second (in reality, the time taken is much lesser). Let us have two iPhones side by side - one fuzzing inputs processed by FEET, and the other fuzzing inputs without FEET. Given 100,000 seconds, the first iPhone would have processed 96,670 unique images whereas the second iPhone would have processed 85,100 unique images. We can say the first iPhone "wasted" 3330 seconds out of 100,000 (a ratio of 0.033) whereas the second iPhone "wasted" 14,900 seconds (a ratio of 0.149) or around 4.5 times as much wastage. This gap only widens when we fuzz for longer durations.

(b) **Crash Arrival Rate Comparison** In Table 3, it can be seen that without using FEET, the device encountered the first crash in its first fuzzing job. However, that is the only crash found; hence this crash could be a one-off incident. Remaining jobs do not yield any crashes. This is in contrast with the utilisation of FEET to first determine uniformity and uniqueness of resulting configurations before generating patches. Crashes come only from the 3^{rd} job onwards.

(c) **Number of Crashes**

There are 14 crashes discovered on the Android 4.0.4 device using FEET selected fuzzing configurations.

(d) **Variety of Bugs**

For the GIF file format, there is only one bug discovered by both FEET selected and randomly selected configurations.

Manufac.	**Version**	**File Format**	**Crash Count**	**#Unique bugs**
HTC	Android 2.3.x	GIF	825	1
HTC	Android 2.3.x	MP4	9	2
Samsung	Android 4.0.x	GIF	253	1
LG	Android 4.0.x	GIF	708	1
Samsung	Android 4.1.x	PNG	19	1
Samsung	Android 4.2.x	MP3	27	2
Samsung	Android 4.3.x	MP3	13	2
Asus	Android 4.4.x	MP4	1	1
Apple	iOS 6.x	JPEG	25	1
Apple	iOS 6.x	JPEG	12	1
Apple	iOS 6.1.x	JPEG	9	1

Table 5: Table of top 10 devices with maximum crashes found using STAMP.

3.3.2 SFAT results

As the result shows in Table 4, the 1^{st} file with higher field and occurance scores generally yield an overall better number of unique crashes and total crashes. We can conclude that SFAT's scoring mechanism is effective for finding a good seed file. The additional crashes are due to a location of the flag change corresponding to the size value of an TBPM (beats per minute) which is not present in 2.mp4.

Seed File	I	Field Occurance Score	Mean/StdDeviation	Size (KB)	#Unique Bugs	#Total Crashes
1.mp4	367.4	77.64	0.2	162	2	8
2.mp4	234.8	33.2	0.66	296	1	1

Table 4: Field score and occurance score for 2 mp4 files. Lesser score is better (See Section 2.1.2)

3.3.3 STAMP results

Within the span of 2 weeks, STAMP conducted close to 5 million test cases and uncovered close to 1900 unique crashing inputs in both Android and iOS devices affecting GIF, JPEG, PNG, MP3 and MP4 video files. The speedup obtained (as shown in Table 6) from distributed parallel fuzzing of 4 devices of the same model achieves a better than expected 5.27. This is likely due to less (or no) crashes happening on some devices in parallel. Network latency may also play a part.

Overall, fuzzing speed depends on the file-type fuzzed. Video and audio files are significantly slower as they have to be played back even though the duration of all seed files are 1 second.

#GIF files fuzzed	#Devices	Time Taken
10000	4	55 min
	1	290 min

Table 6: Speedup obtained with distributed fuzzing Times include downloading, patching and fuzzing.

3.3.4 CACE - Variety of bugs

CACE triaged all discovered crashes discovered into 7 different bugs. They are detailed in Table 5. CACE traced the 1786 GIF crashes to a single bug in the LZW compression algorithm located in the Android 2D graphics library. That crash is an invalid read access and is not likely to be exploitable. On the other hand CACE deduced that the 19 PNG crashes are due to a potentially exploitable write to an invalid memory location also in the same Android 2D graphics library. The MP3 crash indicates an invalid address or address of corrupt block passed to `dlfree` to release memory. Further up the stack, it is actually discovered that the real error happened in the utilities library, where Android was actually trying to free a shared `StringBuffer`. This appears to be a use-after-free vulnerability. However, when `libc` is examined with IDA Pro, `r3` points to the address `0xdeadbaad` [25] intentionally to cause a segment violation and forcibly abort the playback. This is a countermeasure to address a possible heap corruption while rendering the file. The crash for MP3 is due to the TBPM flag set, a tag that is not always used in MP3 files; indicating the importance of fuzzing with a high coverage seed file. The MP4 crashes in Android 2.3.x and 4.4.x are 2 different bugs in the media extractor library. The crash in 2.3.x is likewise a `0xdeadbaad`. The MP4 crash in the newer android version points to a read access violation during media extraction. From the centralised web-console, it we can see that this bug did not affect Samsung devices; cementing our claim that bugs in the Android core system might not affect customised OEM devices and probably vice-versa. The single discovered JPEG crash for iOS 6.1.x is a memory leak bug where an extraordinarily large image dimensions are provided in a much smaller JPEG file. Although the fuzzing yielded no iOS crashes on newer devices. This bug discovered in iOS 6.1.x affects newer iOS devices upon discovering this root cause and re-crafting the JPEG input for them. All bugs have been reported to their respective manufacturers thus the anonymisation of the OS information. We do not yet know if the bugs have been fixed by all affected vendors: Apple Google, OEM and telecom operators around the world. This approach is in line with responsible disclosure best practices.

3.3.5 Storage Utilisation

Table 1 shows the times taken for conducting a 1000 file PNG fuzz job with and without BSDiff.

The process trades storage space for only a minor loss in fuzzing speed as files have to be patched before every fuzzing experiment. A small and good seed file selected by SFAT helps too. On an iOS device, patching a 1000 job packet takes less than a minute. The main disadvantage of using BSDiff patching comes from a lot of CPU time and memory consumed while the patch for a file is produced [23]. This can be easily mitigated when running parallel threads of fuzzing on a high-performance server and running the tests on more instances of mobile devices of the same model.

3.4 Results Discussion

From our experiments, we found that preprocessing fuzzed inputs can be a worthwhile trade-off to make. In Subsection 3.2.2, **spending a few seconds on preprocessing prior to testing reduced the incidence of repeated tests by upto 4 times**. This of course is only a concern when dealing with a large number of test cases as was our case.

Usage of SFAT to maximise RFC coverage has benefits. As explained in Subsection 3.2.1, a file that would not have caused a crash was modified within the constraints of the file format. The resulting file then causes a crash in the target device due to the modified bit. This is just one example among many others and shows that **instead of blind fuzzing, intelligently crafting the inputs beforehand really pays dividends**. On the other hand, automatically downloaded and analysed seed files may not cover 100% of the code coverage as is evident from [14]. Manual insertion of less common fields are still necessary.

FEET is successful in creating fuzz jobs that yield a good number of crashes. It can be augmented by using the Blackbox fuzz scheduling techniques identified in [11] to select jobs based on a smaller epochs and assign weights to determine probability of selection of subsequent jobs after measuring crash ratio obtained from fuzzing one epoch. In this way we could get a crash arrival even faster than by the sole utilization of FEET.

To improve the bug variety discovered, fuzzing operators with a higher entropy such as bit-level operations could provide better results such as with zzuf[26].

The similarity of the number of crashes for multiple devices running Android 4.0.4 devices suggests that the bug

is at the Android AOSP level that is independent of device manufacturer.

Usage of SOFT clients for browser renderable formats are still not very scalable to new platforms. For files renderable in the web browser, the mobile client can be made much simpler.

Exponential storage space growth is also mitigated at some expense of fuzzing speed. However, the payoff can be increased with more users joining the fuzzing experiments.

4. RELATED WORK

The concept of fuzzing was originally introduced in 1990 by Miller et al. [27]. The fuzzer's targets were Unix utilities. Since then a lot of research work has been conducted to improve the methodology of fuzzing. However, mobile Operating systems though Unix based do not expose command line utilities to the average user. With the changing IT landscape new data formats will be defined and existing formats will evolve - all of which require an investment of time and energy to adapt generational fuzzing methodology. Therefore, techniques that use probabilistic heuristics such as mutation ratio, belief metrics and mutation distribution such as our approach are some of the first attempts to provide intelligence to blind fuzzing.

Google [28] has the necessary space and resources to build a fuzzing farm where they have full-control of instrumentation. However, independent researchers and smaller labs have no such luxury. Crowdsourced vulnerability discovery on the other hand has limited control over the remote device.

BFF [29] by CERT is an automated system for finding defects in applications that run on the Linux and Mac OS X platforms. It is also integrated with a tool called CERT Triage Tools that classifies crashes by severity. BFF fuzzes applications which run on the same system as the BFF itself. It is one of the mutational fuzzers that fuzzes a single program with a collection of seeds and a set of mutation ratios. It uses a Multi-Armed-Bandit [32] algorithm to select the best seed-ratio pairs during a fuzz campaign. Due to the shared space, it is easier to coordinate fuzzing, amend fuzzing configuration by feedback and make use of system tools like GDB and Crashwrangler to capture back traces. Also, each fuzzed input is not retained. For testing every new application, the fuzzer is generated and testing cycle repeated. This introduces unnecessary fuzz generation time and duplication of malicious files. There is also no co-relation of fuzz inputs tested across applications or devices that distributed fuzzing provides. Although BFF as a framework is efficient, it is not applicable when the target is a mobile device and runs independently outside the vicinities of the fuzzer without volunteer's administrative access. Automatic Exploit Generation [30] sought to automatically find vulnerabilities in an application and immediately write exploits for them. They have tested the tool on 14 open source projects and discovered 16 exploits. They make use of preconditioned symbolic execution of source and binary code analysis to come up with formula for exploits [30]. This means that they need to have access to the source code. In the mobile domain we know this is not always possible. The exact source code of the Android version in common devices is not released as they contain proprietary code, added on by the OEMs. This is even more true on iOS. Even if the source code does become accessible, the large program size poses a limitation on the extent of manual analysis possible. Additionally, with Mobile OS's implementing Address Space Randomizations (ASLR), symbolic execution methodologies are not effective.

The approach described in this paper for selection of good fuzzing configuration is analogous to the work by both Woo et. al. [11] and Householder et. al. [21]. The former studied how to find the greatest number of unique bugs in a fuzz campaign. They particularly developed an analytic framework using a mathematical model of black-box mutational fuzzing, which modeled black-box mutational fuzzing as a WCCP process with unknown weights and used the condition in the No Free Lunch theorem to decide a better online algorithm. The latter described a workflow for black-box fuzz testing and an algorithm for selecting fuzz parameters to maximize the number of unique application errors. They presented an implementation of the algorithm, which was used to find several previously unknown security vulnerabilities.

Robert et. al [11] have devised a general fuzz confguration scheduling problem-model for selection of seed files for fuzzing independent of the fuzzing scheduling algorithm. In our work on fuzzing, fuzz scheduling is mainly based on the knowledge of position modification.

The authors Woo et. al [8] identified and tested several belief metrics based on past fuzzing results to select the next best seed/program pairs for fuzzing the following campaign. Like our approach, it uses no information of the target and the file format.

We can augment FEET by adopting CERT, Robert's[11] and Woo's[8] approach so a crash can be expected sooner.

To summarize, fuzz configuration scheduling algorithms and belief metrics can be used in conjunction with the selection of modification positioning to improve the number of unique bugs discovered. In particular, the modified positions can be a parameter to select different seed files or fuzz jobs consisting of a disparate set of modification positions based on previously performed experiments.

None of these related work however, address problems associated with fuzzing in a distributed client-server fashion such as pre-generation of fuzz input and reduction of storage overhead.

5. FUTURE WORK

This fuzzing paradigm works against downloadable content which includes documents and multimedia formats. All of which are vulnerable to today's drive-by download attacks. However, system level components also involve network protocols and USB I/O protocols which speak directly to the operating system. Voluntary fuzzing should also expand to these targets as a future work.

MVDP uses byte or block level mutation operators. These operators can only offer overly coarse grained modifications that may be specialised to certain control elements of data formats. Future work can involve the exploration of more mutation operators that operate at the bit level, synonymous to the zzuf [26] fuzzer used by BFF.

We are also working on a specially crafted gateway appliance that can periodically fetch new crash information from STAMP. This information can include modification positions of files that successfully cause an exploitable crash in a particular target device. The modification information can be mapped to the initial/seed file to infer violations of data-format specification predicates. Subsequent incoming

data-streams that match the set of failed predicates will be rejected by the gateway appliance before it reaches the end-host. This application is similar to anti-virus apps on devices, but it checks incoming files rather than applications. The application could also be made extensible by providing hooks into which other sources of information feed known attack signatures.

6. CONCLUSIONS

MVDP was designed to overcome the constraints internet-scale voluntary fuzzing. We used FEET to ensure the fuzzed files are highly unique and uniformly distributed. We designed a method to ensure that the seed files used for fuzzing are of high quality and RFC coverage. We also developed FEET to inform us of the best fuzzing configurations. STAMP and SOFT applications are designed to enable fuzz distribution and testing of the mobile device respectively. The CACE tool enabled automatic binning of unique vulnerabilities from crashes obtained from fuzz testing. Analysis of these crashes and visualizing them according to uniqueness, crash occurrence and severity give us a better chance at exploit generation for the sake of mobile security.

7. REFERENCES

[1] A. Imran, Chronic Dev Team Announces "Tool of Mass Exploitation", Install It Now To Help Community Find Exploits For Untethered Jailbreak *redmonpie.com*, November 27, 2011.

[2] J. Drake, Reversing and Auditing Android's Propietary Bits *RECon*, June, 2013.

[3] Michael Sutton, Adam Greene, and Pedram Amini. 2007. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional.

[4] Sophos Press Release: Users Weighed Down by Multiple Gadgets and Mobile Devices, New Sophos Survey Reveals March 18 2013, Sophos Ltd.

[5] National Cyber Awareness System - Vulnerability Summary for CVE-2012-0003 `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0003`

[6] National Cyber Awareness System - Vulnerability Summary for CVE-2013-0976: `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0976`

[7] National Cyber Awareness System - Vulnerability Summary for CVE-2013-1750 `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1750`

[8] A.Rebert, S.K.Cha, T.Avgerinos, J.Foote, D.Warren, G.Grieco, D.Brumley. Optimising Seed Selection for fuzzing In Proc. 23rd USENIX Security Symposium, 2014.

[9] Graphics Interchange Format, Version 89a, W3C; 31 July 1990.

[10] H.C.Kim, Y.H.Choi, D.H.Lee. Efficient file fuzz testing using automated analysis of binary file format. *Journal of Systems Architecture-Embedded Systems Design*, vol. 57, no. 3, pages 259-268, 2011.

[11] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13). ACM, New York, NY, USA, 511-522.

[12] Open-source software for volunteer computing and grid computing. https://boinc.berkeley.edu/

[13] SETI@home http://setiathome.ssl.berkeley.edu/

[14] C. Miller. How smart is intelligent fuzzing or - How stupid is dumb fuzzing? Independent Security Evaluators, August 3, 2007.

[15] Google Custom Search Engine. https://www.google.com/cse/

[16] Bing Search API. http://datamarket.azure.com/dataset/bing/search

[17] Hachoir Project. https://pypi.python.org/pypi/hachoir-core

[18] Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E) W3C Recommendation 10 November 2003.

[19] Skia 2D graphics library - https://code.google.com/p/skia/

[20] Apple iOS `ImageIO` - `https://developer.apple.com/library/ios/documentation/GraphicsImaging/Conceptual/ImageIOGuide`

[21] A. D. Householder and J. M. Foote. Probability-Based Parameter Selection for Black-Box Fuzz Testing. Technical Report August, CERT, 2012

[22] !exploitable http://msecdbg.codeplex.com/

[23] Binary Diff Utility FreeBSD Man Pages

[24] Mac Developer Library: Apple Technical Note TN233, Accessing CrashWrangler to analyze crashes for security implications, March 2014

[25] (SIGSEGV), fault addr deadbaad https://groups.google.com/forum/#!topic/android-ndk/jQg6DM6-D6o

[26] C. Labs. zzuf: multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

[27] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32(44), 1990.

[28] Chris Evans, Matt Moore and Tavis Ormandy, Google Security Team: Fuzzing at scale http://googleonlinesecurity.blogspot.sg/2011/08/fuzzing-at-scale.html Friday, August 12, 2011

[29] Basic Fuzzing Framework. http://www.cert.org/vulnerability-analysis/tools/bff.cfm

[30] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. Commun. ACM 57, 2 (February 2014), 74-84.

[31] Hex-Rays IDA. https://www.hex-rays.com/products/ida/

[32] D. A. Berry and B. Fristedt. Bandit Problems:Sequential Allocation of Experiments. *Chapman and Hall*, 1985.