Automatic Creation of Computer Forensic Test Images

Hannu Visti(✉), Sean Tohill, and Paul Douglas

University of Westminster, London, UK
{h.visti,s.tohill,douglap}@westminster.ac.uk
http://www.westminster.ac.uk

Abstract. This paper investigates the possibilities for the automatic creation of scenario-based test file images for computer forensics testing purposes, and goes on to discuss and review a tool developed for this task. The tool creates NTFS images based on user-selectable data hiding and timeline management. In this paper we document both the creation of the tool and report on its use in a variety of test situations.

Keywords: Computer forensics · NTFS · Data hiding · File system · Timeline management

1 Introduction

Documented computer forensic test images are necessary to improve the learning process, test students and to test and compare computer forensic software. An undocumented image file cannot be used to verify tool performance or student progress; it is entirely possible that the researcher or teacher is not aware of the full contents of the image. The need for documented test material is well documented. Guo [1], for example, says *However, the growth in the field has created a demand for new software (or increased functionality to existing software) and a means to verify that this software is truly forensic, i.e. capable of meeting the requirements of the trier of fact.*

In Computer Forensic education, test material is needed for both practice and assessed tests. Hands-on excercises have been found to increase interest in the topic and enhance learning results [2], while Agarwal and Karahanna argue *Training programs can also be designed to provide potential users with opportunities for cognitive absorption. Game-based training environments are and more likely to result in cognitive absorption, thus amplifying both beliefs about the instrumentality of the technology and its ease of use, as well as enhancing its adoption and diffusion throughout the organization* [3]. Providing students with "interesting" examination cases while preventing plagiarism requires test images that are random representations of a scenario. All such images should bear equal complexity, to make results comparable if used in testing.

Generally, two different approaches exist in obtaining documented test images. One is to rely on the work of others and the other is manual creation. A few documented test images exist, for example http://dftt.sourceforge.net [4] but testing of

U. Garain and F. Shafait (Eds.): IWCF 2012 and 2014, LNCS 8915, pp. 163–175, 2015.
DOI: 10.1007/978-3-319-20125-2_14

highly specialised tools generally requires manual test image creation. Manual creation, while reliable, can be a cumbersome task if hundreds or thousands of images are needed. This paper introduces a versatile and extensible tool to rapidly create a large number of documented file system images based on a *scenario* defined by the user. All created images based on a scenario fulfill the chosen scenario criteria but contain random elements, thus they are different from each other.

2 Background

2.1 Related Work

A tool (Forensig^2) exists to provide automatic test image creation [5]. Forensig^2 is a versatile tool based on a scripting language to allow users to create images inside a virtual machine. This allows virtually unlimited versatility but does not provide a user interface [6,7].

The strength of Forensig^2 is in its method of creating images inside virtual machines. Virtual machine clocks can be set to any date and time and actions to the image are performed by underlying operating system commands. This makes timeline and contents management easy as long as the user is able to execute all actions in a correct order. If the order of actions is incorrect, this could cause *contamination* to the created images.

Data hiding in NTFS has been studied by Huebner *et al.* [8]. This paper documents a piece of software that implements some of their methods as a proof of concept. The purpose of this software is to create a framework to implement various data hiding methods. The framework is intended to be flexible enough that it will allow the addition of further data hiding methods in the future.

2.2 NTFS

NTFS is a proprietary file system developed by Microsoft. The greatest challenge posed by NTFS is the fact that its documentation has never been officially published [9]. Variance between NTFS versions has also been detected [10]. This paper describes only relevant NTFS concepts.

Master File Table. NTFS contains a file named $Mft, which contains a record for every file in the file system. This is called the Master File Table (MFT) Each *MFT record* contains *attributes*, which in turn contain timestamp information, file storage information, permissions information, and so forth. The first 16 entries are considered system files. MFT record number 0 points to $Mft itself. Record 1 ($MftMirr) is a file containing backup storage of the first 16 entries of $Mft.

Attributes. Attributes can be added to NTFS file systems without losing backward compatibility. However, forensic interest focusses on the following attributes:

– $STANDARD_INFORMATION: This attribute contains timestamp information and is mandatory.
– $FILE_NAME: Stores information about file name and size, and another set of timestamp information.
– $DATA: The actual contents of a file, or in the case of large files, a collection of pointers to the actual contents.
– $INDEX_ROOT, $INDEX_ALLOCATION: B-tree indices.

Directory Trees. While every file could be located from $Mft, a sequential search of this file would be inefficient. NTFS stores contents of directories in B-trees, where structures formatted as $FILE_NAME -attributes contain both file name and a set of time stamps.

2.3 Timestamp Management

NTFS has MACE (modified, accessed, created, entry changed) timestamp information stored in three full MACE sets:

1. $Mft record $STANDARD_INFORMATION attribute
2. $Mft record $FILE_NAME attribute
3. Directory entry $FILE_NAME attribute

Timestamp behaviour inside $Mft record has been studied by Bang *et al.* [11]. Depending on Windows version and action performed to the file, changes are made to only $STANDARD_INFORMATION or to both attributes. Directory entry timestamp information follows $Mft record $STANDARD_INFORMATION attribute despite being in $FILE_NAME format.

3 Design

The tool takes instructions from the user in form of database entries, with a provided user interface. Its outputs are created images and information sheets explaining the contents of each created image. This is illustrated in Fig. 1.

3.1 Terminology

The following definitions are used to describe the image creation process:

– **ForGe:** Computer Forensic Test Image Generator - the tool described by this paper.
– **Case:** Top-level entity. A case contains file system selection, the number of images required, image size, file system parameters, root directory timestamps and timeline variance boundaries.
– **Image:** Images are representations of *cases* and created by the tool.
– **Hiding method:** A method of writing a *secret file* to an *image*.

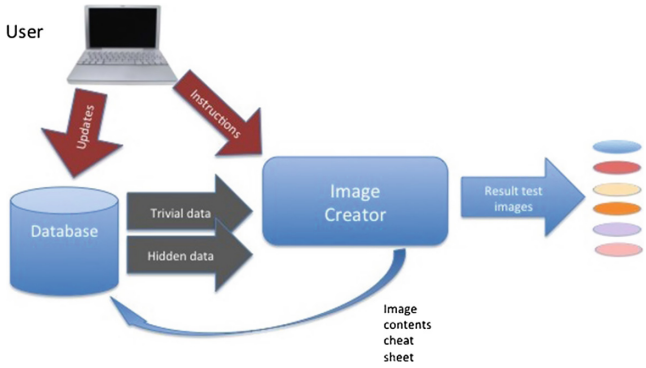

Fig. 1. Inputs and outputs of forensic test image generator

- **Trivial file:** A file without any importance to the *case*. Trivial files are used in file system creation to place files on *images* according to a *trivial strategy*.
- **Trivial strategy:** An instruction related to *case* to place a random selection of *trivial files* in the image.
- **Secret file:** An important file to the *case*.
- **Secret strategy:** An instruction related to *case* to place a single *secret file* using a *hiding method* to an *image*.
- **Action:** A simulated operation performed on a *hidden file*, related to timeline management. For example action "rename" with a timestamp sets the timestamps to correspond to a rename action at the given time.

3.2 Requirements

Top level functionality requirements are outlined in Table 1. The assumption is to build the tool on a Linux platform and use the Linux operating system tools and interfaces where possible.

3.3 Avoiding Contamination

Contamination occurs if a later action modifies or overwrites scenario related information or processes the file system only partially. The following examples illustrate the problem:

1. A file has been placed on the image and its timestamps changed to reflect the scenario. A hiding method then concatenates a secret file to this file. This action changes M and A timestamps to image creation time.
2. A file has been hidden in the file slack of a trivial file. The trivial file is extended, which causes the hidden file to be overwritten.
3. A file has been hidden in the file slack of a trivial file. The trivial file is then deleted. The secret file survives the deletion but if more files are written to the image, these writes can overwrite the hidden file by reusing the now—free clusters.

Table 1. Top-level functional requirements

ID	Requirement	Reason
1	The tool must create working and mountable file systems	As long as file systems are fully functional, it does not limit the choice of forensic tools used to analyse them.
2	The tool must be extensible with new file systems and hiding methods	By creating a framework to manipulate file systems, users can define their own "hiding methods" if those provided are not suitable.
3	A user interface must be provided	A web browser based user interface for database management and application access reduces the expertise needed to operate the tool.
4	The tool must allow the addition of random elements to individual images	If several images are required for external tool evaluation or academic setup, adding the random element to the generator reduces the amount of user work needed.
5	The tool must be designed to be non-sequential in image building. The order of entries and actions configured by the user should not dictate the order in which they are implemented on images	The generator must choose the correct order of implementation to avoid contamination.
6	If an image fails to be created, it must be deleted and the user must be notified	Partial images do not represent the case and must not be allowed to mix with successful images.
7	Successful image contents must be documented and a means of accessing the documentation provided	If exact information of the contents of an image is required, this must be provided in a user-friendly manner. If image generation uses random elements, documenting the actual result for each image is useful if images are used, for example, to assess students.

4. A file has been hidden in empty space. Another file is hidden with the same method. If safeguards are not in place, the second file can overwrite the first file fully or partially.
5. Root directory (MFT entry 5) timestamps are changed. If this change is not reflected in $MftMirr that contains a copy of first 16 file entries, the file system is in an inconsistent state and if mounted, the disk repair process can cause

undesired changes. The same problem occurs if the cluster allocation state is changed and the change is not applied to $MftMirr.
6. When changing timestamps, the three independet timestamp sets are not treated in a coherent manner. File system checks can detect an anomaly and the result might not correspond to the scenario anymore.
7. If new files are written to the file system after deletions have been made, metadata information in $Mft or directory indices can be overwritten.
8. If the last file written to the file system is deleted, NTFS implementations can shrink the size of $Mft file. If this happens, there is no reliable way to locate the MFT entry of the deleted file anymore in case its timestamps should need to be modified.

This list is not exhaustive. Contamination avoidance is a key topic in application design. Instead of treating the user scenario input as a serial course of actions, the image creator needs to understand consequences of performing these actions, and schedule actions accordingly. It is thus essential to know if a data hiding method modifies a mounted file system through operating system commands or writes to raw disk space directly. More volatile actions should be performed after persistent actions.

Modifying timestamp information requires raw disk access. Timestamps cannot be modified using operating system commands without contaminating the result. System calls exist to modify information, depending on NTFS implementation. However, modifying timestamp information is an action in itself, causing an update to "entry changed" attribute. The reliable method of modifying timestamps is to make changes to unmounted file systems. Existing tools, for example Timestomp, also tend to ignore the third set of timestamps located in directory entries [12].

The following order of actions creates a coherent NTFS file system that represents the scenario:

1. Create an empty file system
2. Mount the file system
3. Process trivial strategies: create directories and copy files
4. Process secret strategies that write to a mounted file system using operating system tools or programming language library functions
5. Unmount the file system
6. Process secret strategies that require raw file access, for example those using slack space or unallocated space
7. Mount the file system
8. Write a placeholder file to the file system to ensure the file written last is never deleted
9. Process all file deletions
10. Unmount the file system
11. Change all timestamps, including trivial files, hidden files and deleted files, to correspond the scenario, in all three timestamp locations
12. Copy the first 16 MFT entries from $Mft to $MftMirr

3.4 Addition of Random Elements

For the tool to be useful, all images created to represent a scenario must be different, while maintaining the scenario structure. This is achieved by pulling trivial and hidden files from a pool and allowing random elements in the timeline. Trivial files are arranged by their kind, which include pictures, documents, audio, video and executables. The tool categorises files upon drag and drop upload, based on their file name and signature. A trivial strategy could, for example, be an instruction to create directory /holiday, set date to 10/02/2013 and place 10-20 picture files to the directory. If the user has uploaded 200 picture files to the repository, each image used would be a random selection chosen from this pool.

The secret file pool is not based on file type but grouping. Each secret strategy hides exactly one file from a chosen group using the chosen method. Groups are numeric values assigned to secret files. If a group has exactly one file, each image contains this file. If a group contains several files, one is chosen at random. Secret strategy implementations choose a destination file, directory or raw image location randomly. If, for example, the hiding method "file slack" is being used, the target file, the slack of which is used, is selected randomly from existing trivial files already placed on the image.

Timeline randomisation is based on the addition of full weeks. Each trivial strategy and secret strategy contains timeline information and this is used as point zero in time. If a scenario allows a variance of a maximum of 30 weeks, a random number is chosen for each image and added to every timestamp henceforth. Weeks are used to preserve day of week and time of day information. In an educational setting, an example case might include suspicious activity happening outside working hours or during a weekend. This would be preserved despite providing a different timeline for each image to discourage plagiarism.

4 Implementation

The tool was implemented in Ubuntu 12.04 with Python 2.7.5. Its core processor implements the user interface, the correct order of actions, the handling of trivial strategies, random selection of hidden files, database management and timeline randomisation, while all file system operations and data hiding methods are executed in modules loaded dynamically during execution based on scenarion needs and database information. User interface was created with Django 1.5.1.

4.1 File Systems

File system modules need to implement a documented interface. To achieve this, the file system interface must be able to fully read and parse file system data structures on a raw image file. A limited write functionality is needed to write back timestamp information and handle slack space writing. The following list is an example of file system interface methods:

- fs_init(): reads in file system structures. Can be empty but the method must be present.
- mount_image(): mounts an image.
- dismount_image(): dismounts an image.
- get_list_of_files(flag): returns a list of files that have a certain flag (for example regular file, system file or directory) set.
- find_file_by_path(filename): locates a certain file and returns its data structures
- change_time(path, timedictionary): changes one or more time attributes of a file
- write_location(position, data): writes to raw image space

Linux is dependent on the Tuxera-3G NTFS driver [13]. Tuxera versions prior to 2013.1.13 destroy $FILE_NAME attributes when deleting files, which is not a typical file deletion behaviour in Windows systems [7, p. 41]. The requirement is thus to use Tuxera version 2013.1.13 or newer.

4.2 Hiding Methods

A data hiding method module needs to implement a class with a single method hide_file(file, parameterblock). Data hiding methods return a dictionary containing a human readable string to be inserted into the database, to facilitate displaying image contents. An example of this would be *"File foo.txt hidden in file slack, position nnnnn length zzzz"*. The dictionary can also contain instructions to the core processor if the hiding method requires either timestamp information modifications or file deletions. An example of a hiding method that "hides" a file by deleting it, is provided:

```
def hide_file(self, hfile, param = {}):
    hf = None
    dirs1 = self.fs.get_list_of_files(FLAG_DIRECTORY | FLAG_REGULAR)
    dirs2 = self.fs.get_list_of_files(FLAG_DIRECTORY |  FLAG_SYSTEM)
    dirs = dirs1+[dirs2[0]]
    try:
        dpr = choice(dirs)
        if dpr.filename != ''/.'':
            targetdir=dpr.filename
        else:
            targetdir=''''
        internalpath = targetdir + ''/'' + os.path.basename(hfile.name)
        targetfile = self.fs.fs_mountpoint + internalpath
        hf = internalpath
    except IndexError:
        raise ForensicError(''No directory for hiding'')
```

The above code snippet is a typical starting point for any hiding method. It needs to write the hidden file to the filesystem, and the first task is to find a directory where the file should be written. The list of existing directories is queried from the file system module and one is then chosen by random by the *choice()* function. If the file system is empty and no directories can be found, an exception is raised.

The rest of the method handles the actual file processing and return value generation:

```
try:
    if self.fs.mount_image() != 0:
        raise ForensicError(''Mount failed'')
    tf = open(targetfile, ''w'')
    tf.write(hfile.read())
    tf.close()
    if self.fs.dismount_image() != 0:
        raise ForensicError("Dismount failed")
except IOError:
    errlog(''cannot write file'')
    raise ForensicError(''Cannot write file'')
if internalpath.find(''/./'') == 0:
    internalpath = internalpath[2:]
return dict(instruction=hf, path=internalpath,
        todelete=[targetfile])
```

The file is initially written to the file system and exceptions caught. An exception could be raised, for example, if the file does not fit into the image. The file system is mounted using file system class methods but the actual writing of a regular file is done with Python file operations. The results dictionary contains the file name as a reference to be inserted to database, but also the instruction that the file should be deleted. The hiding method cannot do the deletion itself, as doing it at this stage could cause contamination. Due to the modular structure and strong supporting functionality from the core processor and file system module, actual data hiding methods require little programming.

ForGe implements various data hiding methods that can be considered stable [8]. The selected methods are *deleted file, extension change, alternate data streams, concatenation of files, file slack, steganography* and *unallocated space*. A hiding method of "not hidden" is provided to allow placement of scenario related files into plain sight, with full control over their timelines. *ForGe* is a proof of concept and different data hiding methods were chosen to highlight versatility. For example, file extension change and alternate data streams modify files or file metadata, steganography uses an external open source tool (steghide) and file slack modifies disk space directly instead of with file system tools. *ForGe* offers a framework to do this. It provides access to image files, file metadata and raw disk space. For example in steganography, the framework is used to randomly choose a target file, and to raise an exception if no such files exist, and finally reset timeline in such a way that the modified file cannot be spotted by timeline analysis (Fig. 2).

5 Example

A simple example is provided as a proof of concept. A case of 5 NTFS images is created, with image size of 10 megabytes and cluster size 8. Timestamp variance is set to 26 weeks. Two trivial strategies are created: one to set up */holiday* with 3-6 pictures, and */PDF* to contain exactly two documents.

Trivial strategies provide two directories with random irrelevant files (Fig. 3) as the foundation of images. Three secret strategies are created as illustrated in Fig. 4. The first hides a file by changing its extension, the second places it in unallocated space and the third creates an alternate data stream. Extension

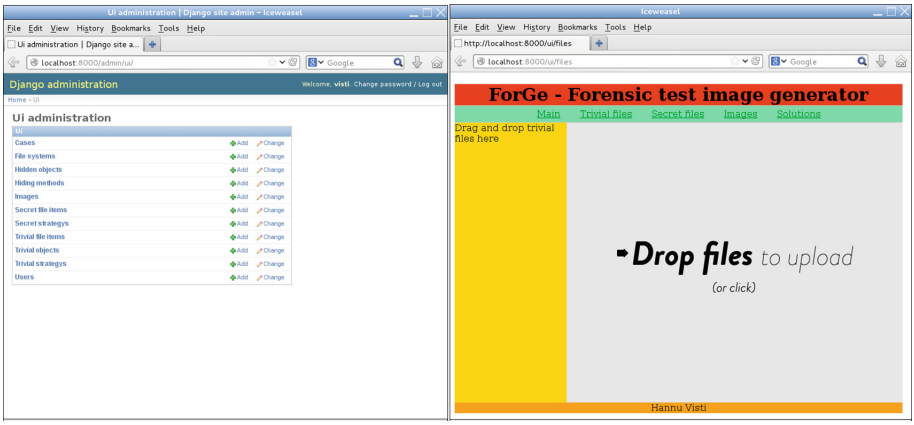

Fig. 2. Main user interfaces

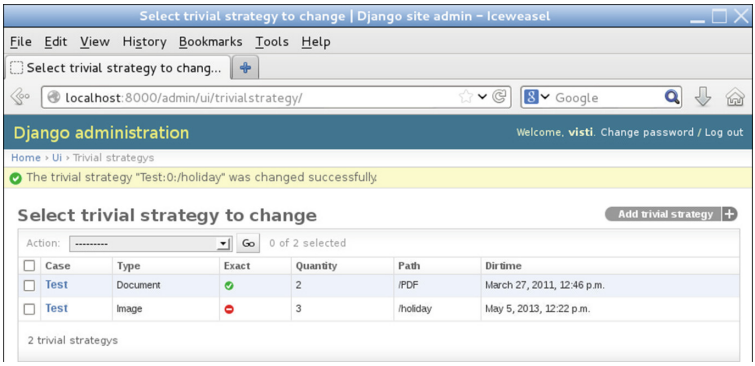

Fig. 3. Trivial strategies of the example case

change and alternate data streams also utilise optional instructions. The extension change hiding method has an option to exclude the root directory from hiding directory candidates. It also has an option to delete the file afterwards. Alternate data streams allows defining the stream name to override the default name "ads".

An example result sheet (Fig. 5) displays contents of the file system and an exact location and timestamp (if relevant) of each hidden file. A simple file extraction test to images 1 (result sheet not included) and 2 demonstrate the result.

Istat displays the NTFS attributes in human readable format (Fig. 6). Timestamps correspond to result sheets and the named $DATA attribute confirm the presence of an alternate data stream. Stream name "foo" is displayed as the attribute name, which corresponds to NTFS alternate data streams functionality.

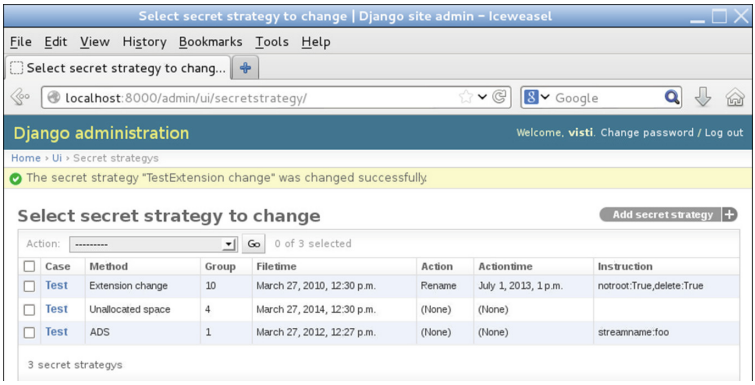

Fig. 4. Secret strategies of the example case

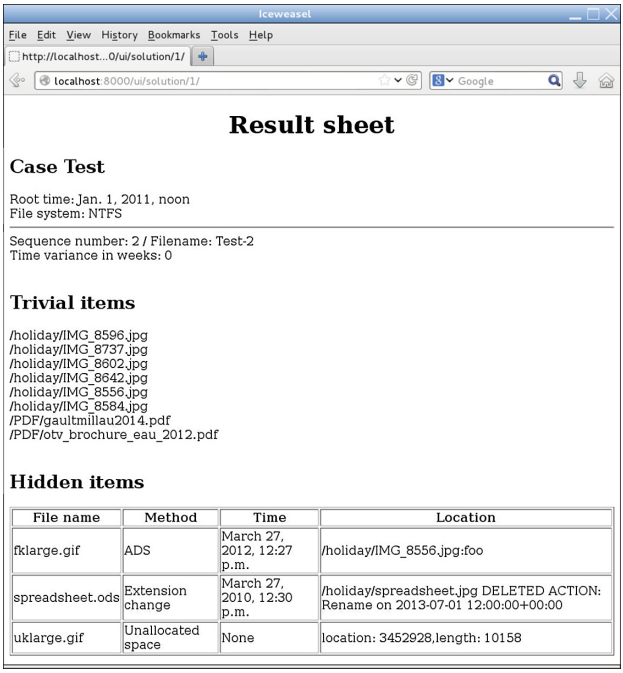

Fig. 5. Contents report of an example image

Creating this case requires only six database entries: One for case, two for trivial strategies and three for secret strategies. Image creation and result sheet display are functions of the *ForGe* application; SQL database access is not needed to analyse results. The only command line interaction would be to copy the created images to a location where they would be used. Access to images is not provided through the user interface.

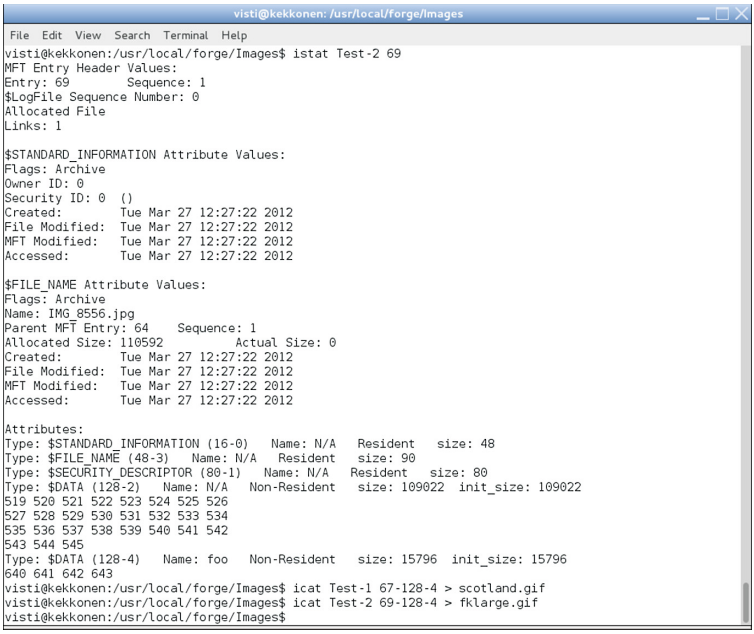

Fig. 6. Timestamps and contents of alternate data streams hiding

6 Conclusions

ForGe is a fast mass image generator with a graphical user interface. It is a fully functional prototype. It lacks some finesse in error handling, and no installation or removal procedures are provided for the program. *ForGe* provides an extensible framework that has built-in mechanisms to avoid contamination and provide full timeline control. It provides information sheets about the images that are created, thus enabling verification of results. Its strength is in rapid mass creation of "similar but not identical" images for forensic software testing or education purposes. Its main weakness is its operation on file and file system level. Higher abstraction level concepts, for example mail folders, web browser caches or backups of mobile devices are not readily supported.

Suggested future work would include removing its current focussing on the placement within an image of individual files, as this would extend its area of usefulness to the creation of, for example, databases and web browser histories.

References

1. Guo, Y., Slay, J., Beckett, J.: Validation and verification of computer forensic software tools-Searching Function. Digital Invest. **6**, S12–S22 (2009)
2. Duffy, K.P., Davis Jr., M.H., Sethi, V.: Demonstrating operating system principles via computer forensics exercises. J. Inf. Syst. Educ. **21**(2), 195–202 (2010)

3. Agarwal, R., Karahanna, E.: Time flies when you're having fun: cognitive absorption and beliefs about information technology usage. mis quarterly **24**(4), 665–694 (2000)
4. Carrier, B.: Digital forensics tool testing images, August 2010. http://dftt.sourceforge.net, Accessed 20/03/2014
5. Moch, C., Freiling, F.C.: The forensic image generator generator (forensig2). In: 2009 Fifth International Conference on IT Security Incident Management and IT Forensics, pp. 78–93. IEEE, September 2009
6. Moch, C.: Der festplatte forensik fall generator, Master's thesis, University of Mannheim (2009)
7. Visti, H.: Automatic creation of computer forensic test images, Master's thesis, University of Westminster (2013)
8. Huebner, E., Bem, D., Wee, C.K.: Data hiding in the NTFS file system. Digital Invest. **3**(4), 211–226 (2006)
9. Carrier, B.: File System Forensic Analysis. Addison-Wesley Professional, Boston, London (2005)
10. Hayes, D., Reddy, V., Qureshi, S.: The impact of microsoft's windows 7 on computer forensics examinations. In: Applications and Technology Conference, pp. 1–6, IEEE, May 2010
11. Bang, J., Yoo, B., Lee, S.: Analysis of changes in file time attributes with file manipulation. Digital Invest. **7**(3), 135–144 (2011)
12. Anon, Timestomp, November 2010
13. Tuxera, NTFS-3G manual (2014). http://www.tuxera.com/community/ntfs-3g-manual, Accessed 20/03/2014