https://doi.org/10.2352/ISSN.2470-1173.2019.5.MWSF-541

Nondestructive ciphertext injection in document files

Scott Craver and Enshirah Altarawneh; Binghamton University, United States
Jugal Shah; Nirma University, Gujarat, India

Abstract

We describe how container files in Portable Document Format (PDF) can be modified to inject modifiable pads for use in a steganographic file system. This produces a footprint that is less conspicuous than a randomized disk volume or disk image file. The PDF standard can be exploited to inject a modifiable segment into a file, without affecting the file's interpretation or validity; these can be disguised as high-entropy segments that commonly occur in PDF files. We show two methods of achieving this embedding.

1 Introduction

The *steganographic filesystem* problem is that of concealing an entire filesystem on a computer, allowing the reading and writing of files to a hidden store whose existence can not be proven. The problem, and a proposed solution, was first articulated by Anderson, Needham and Shamir in [1]. This problem is substantially different from the conventional steganographic problem of embedding a fixed payload in a cover object. There are several elements of this problem that make it unique:

- A steganographic filesystem requires a very large overall payload size, which can challenge capacity limits in conventional steganographic embedding. It also entails a payload that is constantly modified, rather than a fixed message.
- A steganographic filesystem requires an embedding method that meets the performance requirements necessary for file system operation.
- The "cover object" in the steganographic filesystem problem is a computer's disk volume. This is a highly complex object that can not be regarded as a signal, or processed or analyzed using a signal-processing framework as we can with image steganography.
- Conventional steganography aims to achieve statistical undetectability, whereas the steganographic filesystem problem, as originally posed, aims for a weaker goal of *plausible deniability.*

Plausible deniability refers to the security property that a user can reasonably deny a message, in this case denying that files are present in a system, for example that the steganographic filesystem was installed but never used. Practically, this means that the presence of steganography may be established but no determination can be made of the payload size.

The solution in [1] was to randomize the unused blocks in a disk volume, filling a disk with i.i.d. uniform random bits. Data is then hidden on the disk first by encrypting it, and then overwriting disk blocks with the encrypted data. This exploits the fact that strong ciphertext should be computationally indistinguishable from random bits. This method was later implemented in [2]. A software product called *TrueCrypt* could perform a similar concealment of ciphertext within a file of random bits. [4] In so doing, it would be computationally impossible to determine how much data, or if any data, is immersed among the random background. A file system is then achieved using this medium as a logical device.

This method has several advantages. One is that embedding is simple and fast, as it simply consists of writing an encrypted payload on to a disk block or file. Another is that the embedding is theoretically undetectable, because it uses an artificial cover designed to have identical statistics to the embedded data.

1.1 Towards innocuous embedding

The primary problem with the method of embedding described in [1] is that it does possess a clearly conspicuous footprint in the form of a randomized volume. Essentially it uses an artificial cover consisting of i.i.d. random bits, which does not qualify as an innocuous cover object in the prisoners' problem. If Alice and Bob were allowed to send each other files of random bits, they would be able to send each other ciphertext without using steganography at all.

In practice, this means that a steganographic filesystem of this type will betray its existence, enough that a user may then become a target of more focused surveillance. If the motivation of the steganographic file system is to avoid the conspicuous nature of clearly visible encrypted files, then such a system should not have a footprint consisting of clearly visible volumes of random data.

In this paper, we describe how the random segments used in this approach can instead be injected, plausibly, into files commonly found on a computer. This can then produce a disk volume with no suspicious files or randomized blocks, and bring us closer to the goal of an innocuous-looking disk volume.

This alternative type of concealment is part of a larger system called AINT, a steganographic operating environment that combines a file system with a virtual operating system with applications and a desktop user interface. [3]. The paper outlining the AINT system proposed concealment of data in files, but did not detail how this could be convincingly achieved. We outline several effective strategies for concealment in Portable Document Format (PDF) files.

2 Nonintrusive embedding in container file formats

Our goal is manipulating a file in a common format to include a *modifiable pad*: a region of bytes that can be altered to a store a message, without corrupting the file or affecting the operation, display or interpretation of the file. If such a pad is placed in a known location and length, one can embed data in the file and extract data from the file simply by seeking to that location and writing or reading bytes, respectively. A stegnographic file system could then consist of a database that keeps track of modifiable pads in files, distributing data among them.

Instead of modifying a file to contain a specific message payload, we instead consider the problem of modifying a file to contain a random string of bytes, where by random we mean independent and identically distributed (i.i.d.) bytes with a uniform distribution. Embedding will consist of overwriting some or all of those bytes with ciphertext, under the assumption that the ciphertext is computationally indistinguishable from i.i.d. uniform bytes. Hence the detectablility problem is not distinguishing cover objects from stego objects, but distinguishing cover objects to those objects prepared to accommodate future stego data.

To achieve this reliably, we need to meet the following specific constraints:

1. A modified file must remain valid according to its file format, and not cause unwanted behavior in computer programs using those files.
2. The embedded pad should have no effect on the file's behavior, regardless of its byte values.
3. The embedded pad should not be inconspicuous for the file or its format.

We will relax these contraints probabilistically, requiring that they be met with high probability for random choices of the pad string value. Thus there may be conspicuous pad strings (`"STEGO FOUND HERE"`) that are unlikely to occur at random, assuming that embedded data is encrypted before immersion.

2.1 Embedding and extraction from a modifiable pad

We do not want to employ an embedding algorithm that processes or modifies file data with any sophistication, parsing or understanding of its contents. Instead, we wish to create a region of a file where a segment of random data is plausible and inconspicuous, so that random data can be written directly in to that spot.

If this is achieved, barely any software is needed for embedding and detection. Both are, at a rudimentary level, acheivable using standard Unix commands. The following `dd` command will write N bytes from the result of a program into a file at position P:

```
./program | dd of=file bs=1 seek=P
    count=N conv=notrunc
```

To extract the same bytes from a file and pass them to standard output, we write:

```
dd if=file bs=1 skip=P count=N
```

If a computer has an `openssl` command, a command-line chain can encrypt and decrypt data that is embedded or extracted from a file. The AINT system described in [3] exploits this capability by concealing steganographic filesystem software in the filesystem itself; a terminal command chain can be used to extract and execute a "bootstrap" script that launches the remaining software from its hidden location.

This design, however, fixes the architecture of the system so that the data in a modifiable pad must be ciphertext, not modified to resemble the statistics of a file. This is why our embedded pad is assumed to be i.i.d. uniform random bytes, and thus our third constraint requires that a hunk of i.i.d. uniform bytes be inconspicuous when laid in a file.

2.2 Suitable file formats for pad injection

Candidate files for injection of a modifiable pad should have the following properties:

- Such files should be commonplace on personal computers.
- Files should be large enough that the additional space of an injected pad is not conspicuous.
- Files should commonly hold parcels of high-entropy data, so that such a parcel of i.i.d. bytes laid within the file is not conspicuous.
- A file should be easily modifiable to include a parcel that does not impact the behavior or function of a file.
- An ideal file for embedding would be a file that is not likely to be edited or modified, so that an injected pad is not corrupted or moved.

A fruitful approach for embedding is to consider *container files*, for example Portable Document Format (PDF) files, which hold numerous resources such as text, figures, or images. By their nature they should allow the inclusion of additional data parcels, some of which can be of high entropy. In addition, PDF is commonly used as a terminal format for documents, so that they can lie unmodified on a disk. We have focused our embedding efforts on PDF files.

3 Structure of Portable Document Format (PDF) Files

Figure 1 shows the overall structure of PDF files. The file begins with a header section with information such as the version of the pdf file. Thereafter we have a body that contains various objects that encapsulate data for displaying the document. Objects are declared using a text format, although the object data can contain a stream of binary data. Each object has an index number, and can be referenced by other objects, using its index number.

An object can simply be a string or simple parcel of data, with the following format:

```
<Object Number> 0 obj
<data>
endobj
```

A *stream* object begins with a dictionary of name-value pairs, and the keywords `stream` and `endstream`, between which we may have arbitrary data:

```
<Object Number> 0 obj
<<DICTIONARY>>
stream
<stream>
endstream
endobj
```

This type of object is used to store compressed data or image data in a binary stream. The dictionary explains how the stream is supposed to be interpreted; in PDF terminology, the dictionary names *filters* that are supposed to be used when processing the data. For example, the `/FlateDecode` filter is used to decompress a binary stream according to the Flate compression standard.

After the objects the PDF file has a cross-reference table, with a logical byte offset for each object in the file, for rapid access. This table also denotes whether an object is in use by the document, or unchained and free to delete. A final section of the file provides a logical file location of the XREF table, so that a PDF reader can find it by working backwards from the end of the file. This section dictionary contains reference to root object, count of objects, ID of the pdf file, object number of the object containing the information of the creator as well as creation date.

It is possible to appends subsequent objects to a PDF file, followed by a new cross-reference table and trailer. This allows updates without changing the byte offsets of previous objects.

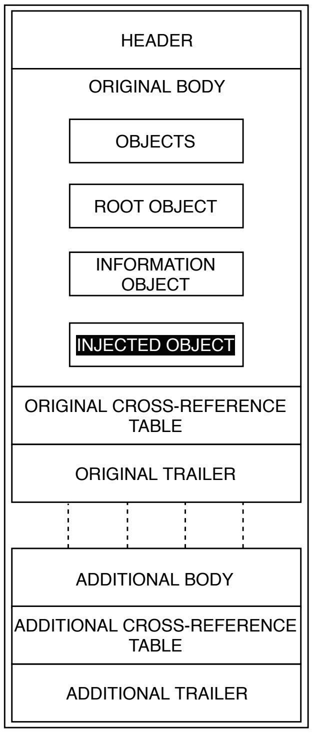


Figure 1: Structure of PDF file
Showing where Object Injected

3.1 Embedding strings in PDF files

The above file format suggests a clear strategy for injecting a string into a PDF file without affecting the file's rendering: one could add an extra object to a PDF file, filled with a stream of arbitrary data. Because the object is not used in rendering, it will not affect the file's interpretation or the document's appearance.

Injecting such an object presents several problems, however. If the injection displaces any existing object, the cross-reference table must be modified. Because objects are indexed and then referenced by their index values, a new object in the middle of the object list will either be conspicuously out of order, or require the reindexing of other objects. These problems can be avoided by placing the new object at the end of the object list, but this fixes a location where the embedded object may be easily spotted. Additionally, an additional unused object may be spotted as unreferenced when a file is rendered.

A second approach is the modification of an existing object to include an injected pad, which we will describe subsequently. Before doing so, we discuss what objects are commonly found in PDF files, that can be used as decoys for an embedded payload.

4 Analysis of PDF file streams

PDF file streams are a strong candidate for embedding binary data, so we assembled a corpus of 2170 PDF files from computers in our laboratory. From these files we extracted all stream objects. Most of these streams were encoded for three compression filters: FlateDecode, DCTDecode, and CCITTFax. It is thus reasonable to create an object disguised as one of these streams.

Figures 2 through 6 show scatter plots of stream sizes, relative to the file sizes. These show that FlateDecode streams are by far the most common, and that their streams are commonly within one to two orders of magnitude of the file size. Because these streams are common and because they are commonly large, they are a suitable cover for placing a large pad in a file.

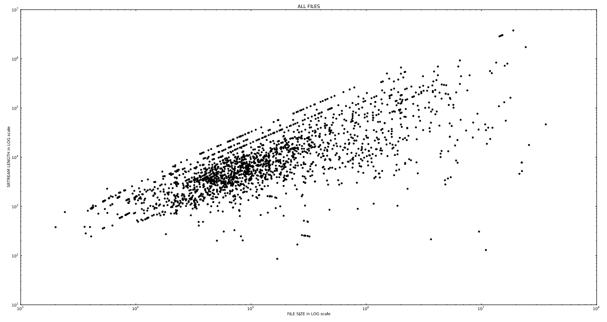


Figure 2: File Size vs Stream Length of Each File

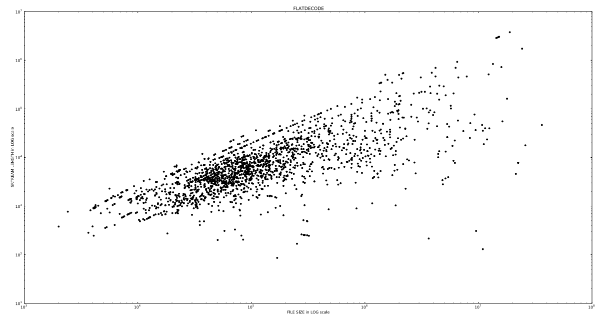


Figure 3: File Size vs Stream Length of Files with FLATDECODE Filter

We also estimated the byte entropy of streams with different filters, as shown in figures 7 through 10. Compressed streams are expected to have high entropy, and a high-entropy segment such as a random string or section of ciphertext can therefore plausibly masquerade as a FlateDecode stream.

Based on these results, we decided that we would embed modifiable pads into a PDF file as a FlateDe-

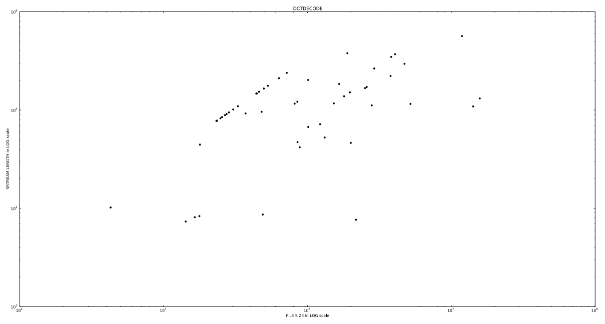


Figure 4: File Size vs Stream Length of Files with DCTDECODE Filter

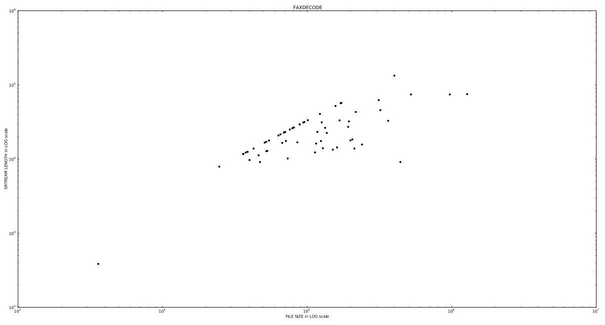


Figure 5: File Size vs Stream Length of Files with CCITTFAXDECODE Filter

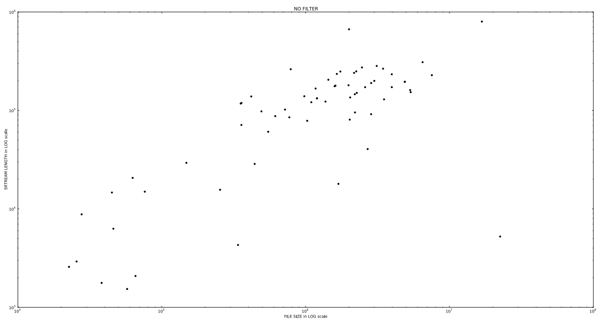


Figure 6: File Size vs Stream Length of Files with NO Filter

code stream object.

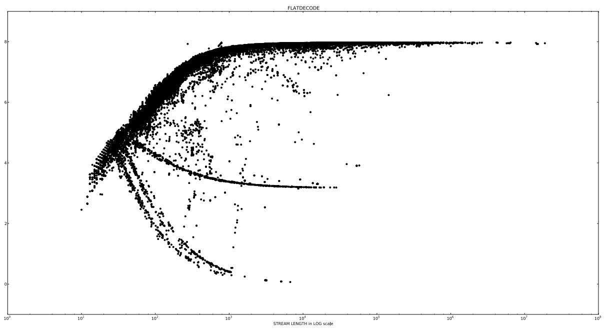


Figure 7: Stream Length vs Entropy of Streams with FlateDecode Filter

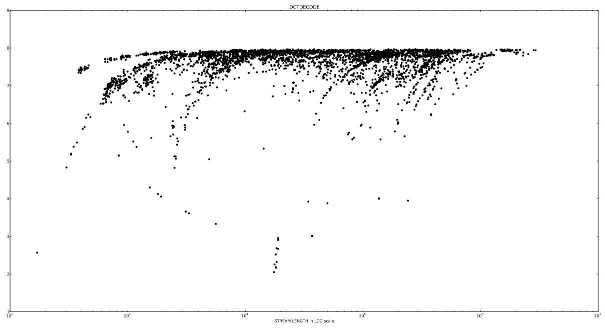


Figure 8: Stream Length vs Entropy of Streams with DctDecode Filter

5 Embedding methods

We employed two approaches to embedding. The first method was to augment a PDF file with a new FlateDecode stream object, whose data is random noise rather than a properly compressed Flate stream. This would cause an error upon decompression, but the object should not be invoked when the object is rendered.

Our second method was to identify a FlateDecode stream object, and then add a length of random data after its stream but before the `endobj` tag, updating the stream length in the object's dictionary. This was based on a careful analysis of the Flate compression format. A proper Flate stream consists of a brief header, a parcel of compressed data in blocks, and a 32-bit checksum of the decompressed data. [5] Each block specifies the type of compression used, the block length, and a flag denoting a terminal block. Any bytes after the terminal block and checksum are not processed by the decompressor, and an arbitrary string can lie beyond them. This does not produce a corrupt file, because the extra bytes are accounted for in the length field of the object dictionary.

5.1 Results

We implemented both methods, testing them on our corpus of PDF files and ensuring that all would display properly on the MacOS Preview application (Preview version 8.1 on MacOS 10.11.6). Thereafter,

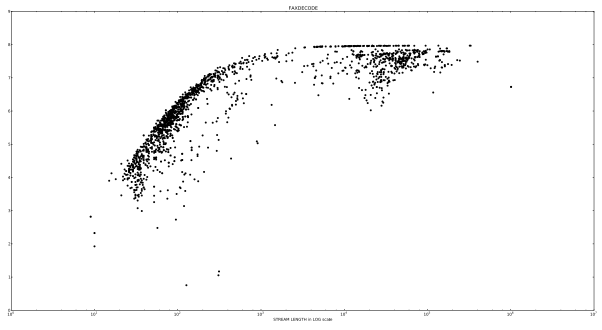


Figure 9: Stream Length vs Entropy of Streams with CCITTFaxDecode Filter

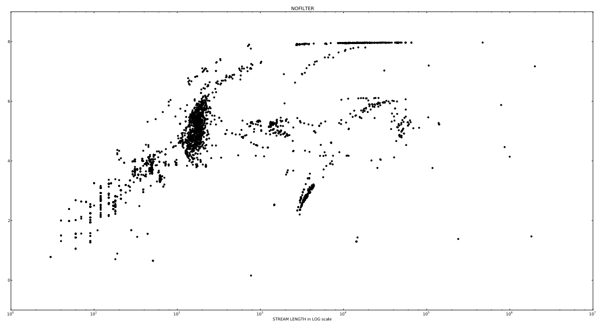


Figure 10: Stream Length vs Entropy of Streams with No Filter

files were tested against several PDF validation utilities. This was not meant to demonstrate the effectiveness of the embedding methods, but rather to ensure that our embedding software had no errors; either method should produce a file that passes validation, if implemented properly.

Our first embedding method was implemented in C, without the use of existing PDF processing libraries or utilities. The end result was not satisfactory, owing to the complexity of adding an extra object to a PDF: we could only place the injected object at the end, rather than arbitrarily within the document, and on several files the resulting file did not pass validation. This we attribute to an error in our code that we could not fix by the time of this publication; we found that we could post-process the afflicted PDF files with the *pdfrw* library, and the modified files passed validation while still containing the embedded pads. [7].

Our second embedding method was implemented in Tcl, using the *qpdf* PDF processing utility. [6] The *qpdf* utility can translate a PDF file into a temporary "QDF" format whose objects can be easily changed in size, which is then re-indexed when converted back to the PDF format. Our program scans a QDF file for a FlateDecode object, appends a pad of a requested length, and then updates the stream length field for that object. Upon conversion, the document displays correctly and passes validation. This was easier and more reliable than the introduction of a new stream object.

Comparing the two methods, we determined that the second method was more reliable, and allows us to embed a modifiable pad wherever FlateDecode objects are found within a document.

5.2 Attacks

However innocuous these injected objects might seem, both of the above methods can be identified by a program designed to search for these modifications. Objects of the first embedding method are easily identified by attempting to decompress all FlateDecode streams in a file. This can be achieved without attempting to render the document: we wrote a Python program that removes all Flate streams, feeding each to the `zlib.decompress()` function of the Python zlib package. The embedded pad is not a valid FlateDecode stream, so this call throws an exception.

The second method produces a valid FlateDecode stream which passes this test, but it has a separate problem: because the modified stream has a suffix of spurious bytes, a byte can be removed from the end and it will still decompress without error. A proper FlateDecode stream should fail to decompress if the last byte is missing, because that would be part of the checksum used by the stream.

It is therefore possible for someone to scan for strings embedded using both of these methods, if either one is known or suspected to be in use.

6 Discussion and conclusion

We outline two methods to inject random strings nondestructively into a PDF document, so that the injected strings can be used as a volume for a steganographic file system. The purpose of this injection is to avoid the conspicuous nature of random volumes or randomized blocks normally associated with these systems, by placing random bytes within files where high-entropy data is plausible.

As described in [3], there are other advantages to concealing a random volume within files. The files are more easily transferred from one computer to another, unlike unused disk blocks, and embedding can be accomplished without requiring administrator privileges on a computer–one only needs permission to modify one's own files.

However, the injection as we performed it can be identified by anomalies in the objects that contain them. It is an open question whether a file format allows the nondestructive inclusion of a random string that can not be spotted by a straightforward forensic test. Nevertheless, this injection is preferable to the approach of earlier steganographic filesystems, which placed such randomized pads conspicuously on a disk.

References

[1] Anderson, R., Needham, R., and Shamir, A., "The steganographic file system," *Lecture Notes in Computer Science* **1525**, 7382 (1998).

[2] McDonald, A. and Kuhn, M., "A steganographic file system for linux," *Lecture Notes in Computer Science* **1768**, 463477 (2000).

[3] Ashendorf, E. and Craver, S., "Design of a steganographic virtual operating system," in *Proceedings of SPIE*, vol 9409 (2015).

[4] Czeskis, A., Hilaire, D. J. S., Koscher, K., Gribble, S. D., Kohno, T., and Schneier, B., "Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling os and applications," in [*Proceedings of the 3rd Conference on Hot Topics in Security*], 7:1–7:7 (2008).

[5] Deutsch, P., "ZLIB Compressed Data Format Specification version 3.3." Internet Request for Comments, RFC 1950, May 1996. https://www.ietf.org/rfc/rfc1950.txt [Online; accessed 28-January-2019].

[6] "QPDF: A Content-Preserving PDF Transformation System." http://qpdf.sourceforge.net/ (2019). [Online; accessed 28-January-2019].

[7] Driscoll, M. "Creating and Manipulating PDFs with pdfrw." https://www.blog.pythonlibrary.org/2018/06/06/creating-and-manipulating-pdfs-with-pdfrw/ [Online; accessed 28-January-2019].

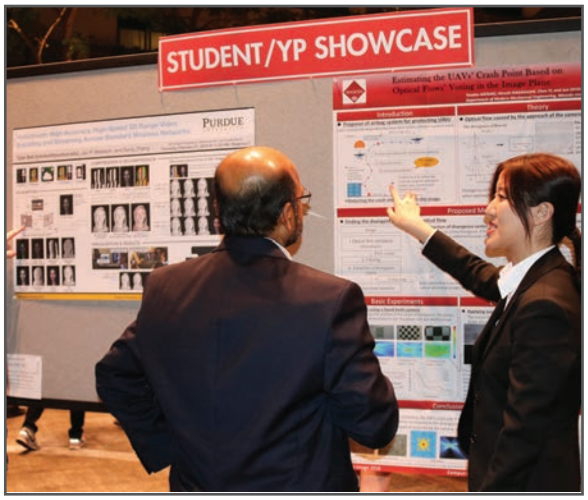
JOIN US AT THE NEXT EI!

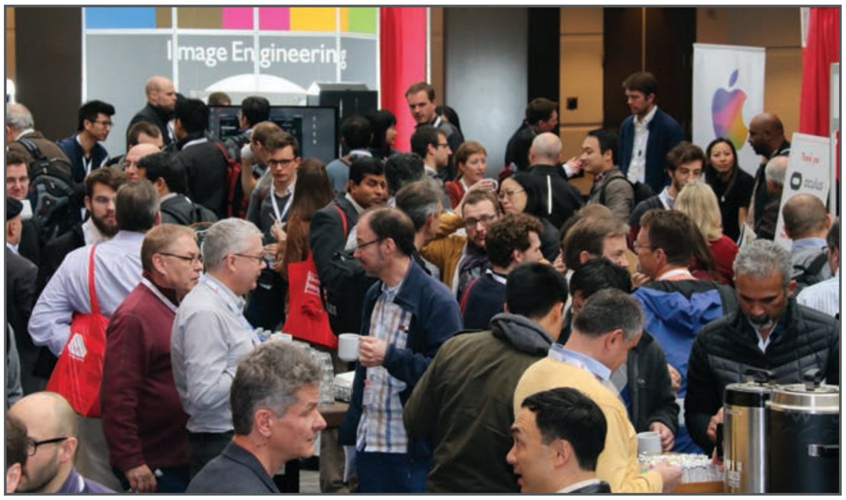
IS&T International Symposium on

Electronic Imaging

SCIENCE AND TECHNOLOGY

Imaging across applications . . . Where industry and academia meet!





AR is a fundamental and transformative interface and platform.
20+ years from my initial working demo to today's AR displays
Optimistic that it won't take another 20 to reach ubiquitous consumer AR!
My first working AR prototype
Today's platforms
Ubiquitous consumer AR

• SHORT COURSES • EXHIBITS • DEMONSTRATION SESSION • PLENARY TALKS •
• INTERACTIVE PAPER SESSION • SPECIAL EVENTS • TECHNICAL SESSIONS •

www.electronicimaging.org

IS&T
imaging.org