A Note on the Confinement Problem

How to prepare a sample talk

Alastair Beresford
January 2020
The paper introduces a data security challenge. How were these framed?
The confinement problem

“The customer will want to ensure that the service cannot access (i.e. read or modify) any of his data except those items to which he explicitly grants access”
Context: computing in the 1970s...
Concern was about leaking confidential data, not correctness.

Can you remember the seven ways of leaking data?
Concern was about leaking confidential data, not correctness

0. Leak via shared memory
1. Write data to a permanent file
2. Write data to a temporary file
3. Send data via IPC
4. Encode data in the bill for service
5. Encode data through write-locks on files
6. Artificially modulate system resource usage
Proposed sol\textsuperscript{n}: use confinement to block explicit sharing

Three properties suggested:
• A confined program must be \textit{memoryless}
• A confined program must make no calls to any other (unconfined) program
• A trustworthy supervisor

What does a trustworthy supervisor need to do?
Side channels and covert channels are described

“Examples 5 and 6 show that it is hard to write a trustworthy supervisor, since some of the paths by which information can leak out from a supervisor are quite subtle and obscure. The remainder of this note argues that it is possible ... It is necessary to enumerate them all and then block each one”
The structure and content of a paper and a talk may differ

Paper structure
• Introduction
• The Problem
• Confinement Rules
• Summary

You don’t need to explain all the detail: focus on the important ideas
What doesn’t the paper talk about?
What doesn’t the paper talk about?

- Programming-language security
- Attacker models
- Cryptography
- Computer networking
- Anonymous users
- Mobile and cyber-physical systems
Was the work novel at the time?


Possible talk structure

- Historical context: who, what, why? 1 min
- Interpreting and explaining terminology 2
- Ideas found in the paper 7
  - The overall challenge
  - Ways data might leak
  - Confinement
- Under explored ideas present at the time 2
- Papers cited and other ideas at the time 1
- Changes which have occurred since publication 2
- In what ways is the work (in)valid today? 2

(total: 17 mins)
Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION
I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX1 swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE 1
In college, before video games, we would amuse ourselves by posing programming exercises. One of our favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in CO programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

1 UNIX is a trademark of AT&T Bell Laboratories.