Key idea:

- An application has a certain normal behavior. And when an attacker attacks the application, it does not behave the normal way.

Behavior:

- System calls made by applications are chosen as the behavior in this paper.
- The reasons behind selecting system calls are:
  1. System calls have more privileges.
  2. Attacker must use system calls to cause any kind of damage to the system.
  3. When the application tries to connect to the OS (through system calls), system calls are the right place to intercept what's going on, rather a convenient point of mediation.
  4. Therefore, if an attacker does not make any system calls, it cannot write to a network, cannot access the database and cannot attack anything.

Normal Behavior:

- Normal behavior is defined as the expected behavior or the simplest model of behavior or the set of possible system calls.

Implementation

- When an application wants to perform a system call, it will check it against a model, say M.
- If the system call is consistent with the model, the operation is allowed.
- Otherwise, it will kill it (depends on what the user wants).

Evaluation:

- The paper is bad at False Negatives and has no false positives.
- There are two ways to construct a model, M
1. Automatically from the source code, the linker brings them all together.
2. Automatic from execution traces (helps customizing according to our script).
3. Manual construction is not possible.
4. Automatically from binaries. This covers libraries also but not the shared ones.

Ngram Model:

- Till now, we are not capturing the order of the system calls anywhere. Therefore a better suggested model is the ngram model.
- \( m = \{(,...),(...),\ldots\} \), a sequence of \( n \) system calls that an application might perform.

Constructing an Ngram Model:

- For the following code,
  ```c
  int main()
  {
    int fd=open(...);
    if (fd>=0)
      read(...);
    else
      write(...);
    fork();
    open(...);
    ioctl(...);
    close();
  }
  ```
- For the above piece of code, the pair of system calls which get executed one after the other are as follows:
  - (open, read)
  - (open, write)
  - (open, ioctl)
  - (write, fork)
  - (fork, open)
  - (read, fork)
  - (ioctl, close)

- This model has a loop in it, namely (open, read), (read, fork) and (fork, open), which implies it shows some weird behavior and therefore more false negatives.

How would the OS check such models?
- Probably by building a state machine.
- The CFG for the above code looks like

```
open()

read()
write()

fork()
ioctl()
close()
```

- So, considering the CFG, assuming that the OS does not know which path to take, the problem of loops can be solved.
- We can build this model by analyzing the source code.
- We cannot build it from the traces of system calls as we may not be able to have some of the states.
- OS checks conformity to the model by remembering the state in which it is and then moving to the next state on the next system call.
- If we have an NFA, we could convert it to a DFA. Also, when you move over an € and you encounter a system call, all you need to do is to check if that system call can be reached over any other path.