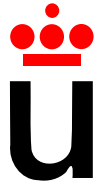
SealFS: logs resistentes a la manipulación

"No me toques los logs"

Enrique Soriano Salvador
enrique.soriano@urjc.es
Gorka Guardiola Múzquiz
gorka.guardiola@urjc.es

GSYC

4 de marzo de 2020



Universidad
Rey Juan Carlos

who

Enrique Soriano Salvador: Doctor en Informática y actualmente Profesor Titular en la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universidad Rey Juan Carlos de Madrid. Sus líneas de investigación se centran en los sistemas operativos, los sistemas distribuidos y la ciberseguridad. Ha participado en la creación de distintos sistemas operativos de investigación (Plan B, Octopus, Nix) y arquitecturas de seguridad. Desde 2002, ha colaborado con distintos grupos de investigación internacionales (Google, Bell Labs, Ericsson, etc.) y ha publicado múltiples trabajos en revistas científicas y congresos internacionales. También ha participado en diversos proyectos de software libre y en la organización de congresos y simposios.
Más información: `https://gsyc.urjc.es/esoriano`

Gorka Guardiola Múzquiz: Profesor Titular en la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universidad Rey Juan Carlos de Madrid. Estudió Ingeniería de Telecomunicación en la Universidad Carlos III y tiene un doctorado en Ingeniería informática en la Universidad Rey Juan Carlos. Ha participado en el desarrollo de los sistemas operativos de investigación Plan B, Octopus, Plan 9 y Nix, trabajando en equipo con programadores de Google, Bell Labs, Sandía National Labs entre otros. Ha publicado múltiples trabajos en revistas científicas y congresos internacionales. Investiga en sistemas operativos, sistemas embebidos, programación concurrente, seguridad y matemáticas puras y programa software libre cuando se lo permiten sus obligaciones.
Mas información: `https://github.com/paurea` y `http://paurea.net`

tl;dr ... UNIX geeks!

El problema

- Caja negra para robot / ordenador
- Ficheros de log fiables

El problema

- ¿Puedo confiar en los logs?
- Un atacante que se haga root los puede manipular

El problema: fases de un ataque

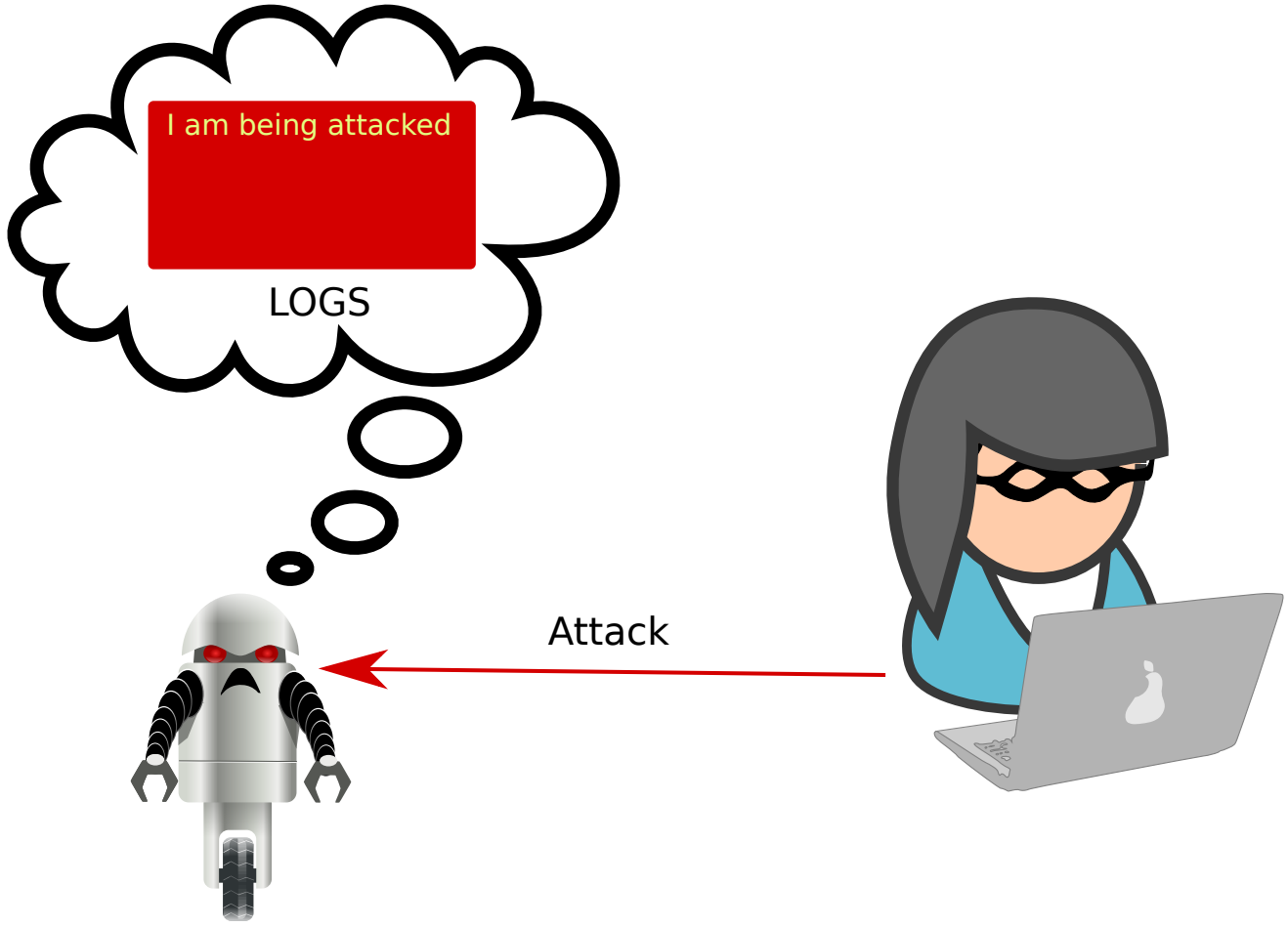
0. Funcionamiento normal, no hay ataque
1. Comienza el ataque
2. Atacante sin privilegios
3. Atacante con privilegios totales

Fase 0: No hay ataque

- Funcionamiento normal
- Los logs guardan información

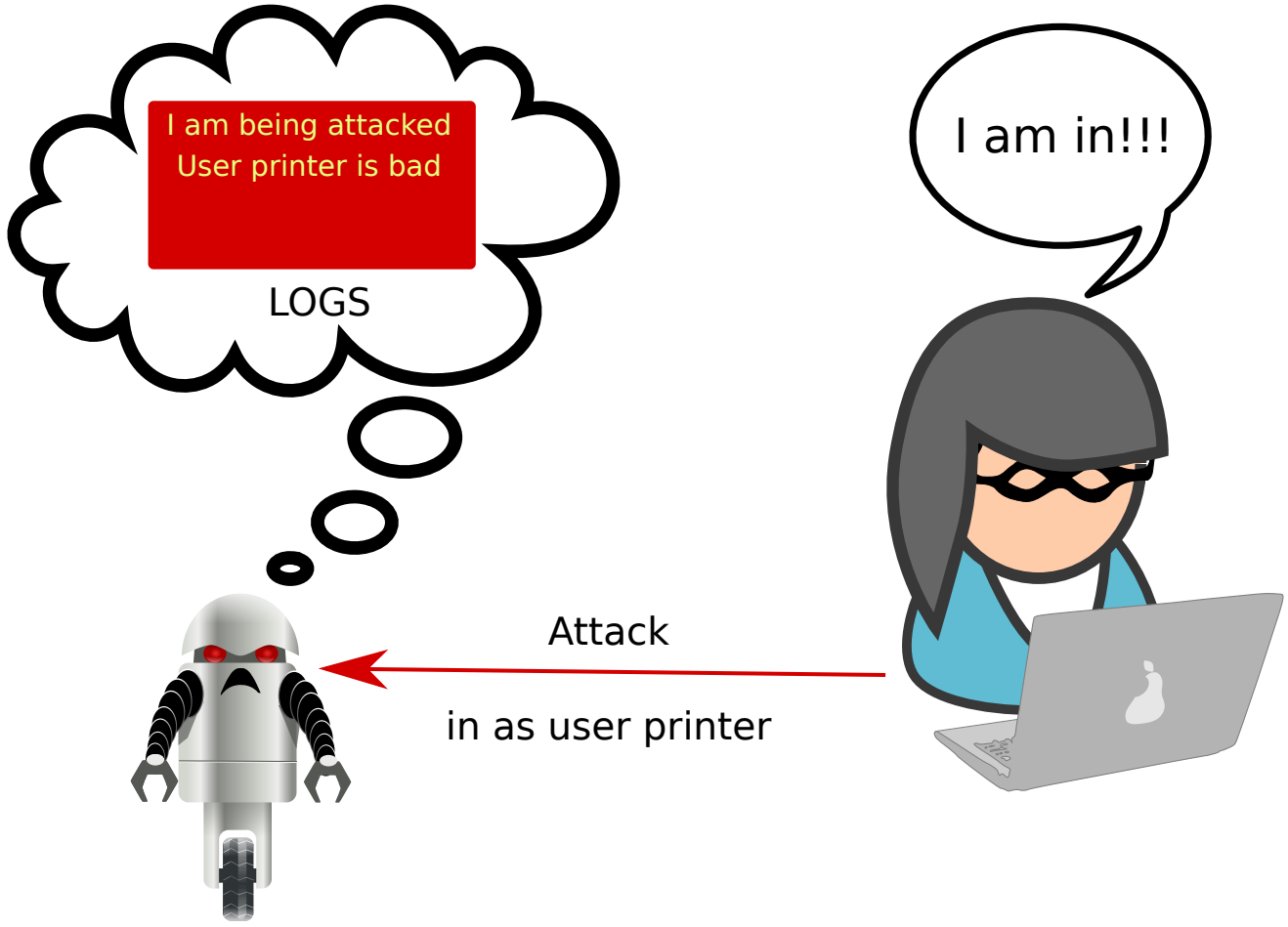
Fase 1: Comienzo del ataque

- Funcionamiento normal
- Atacante intenta ganar acceso



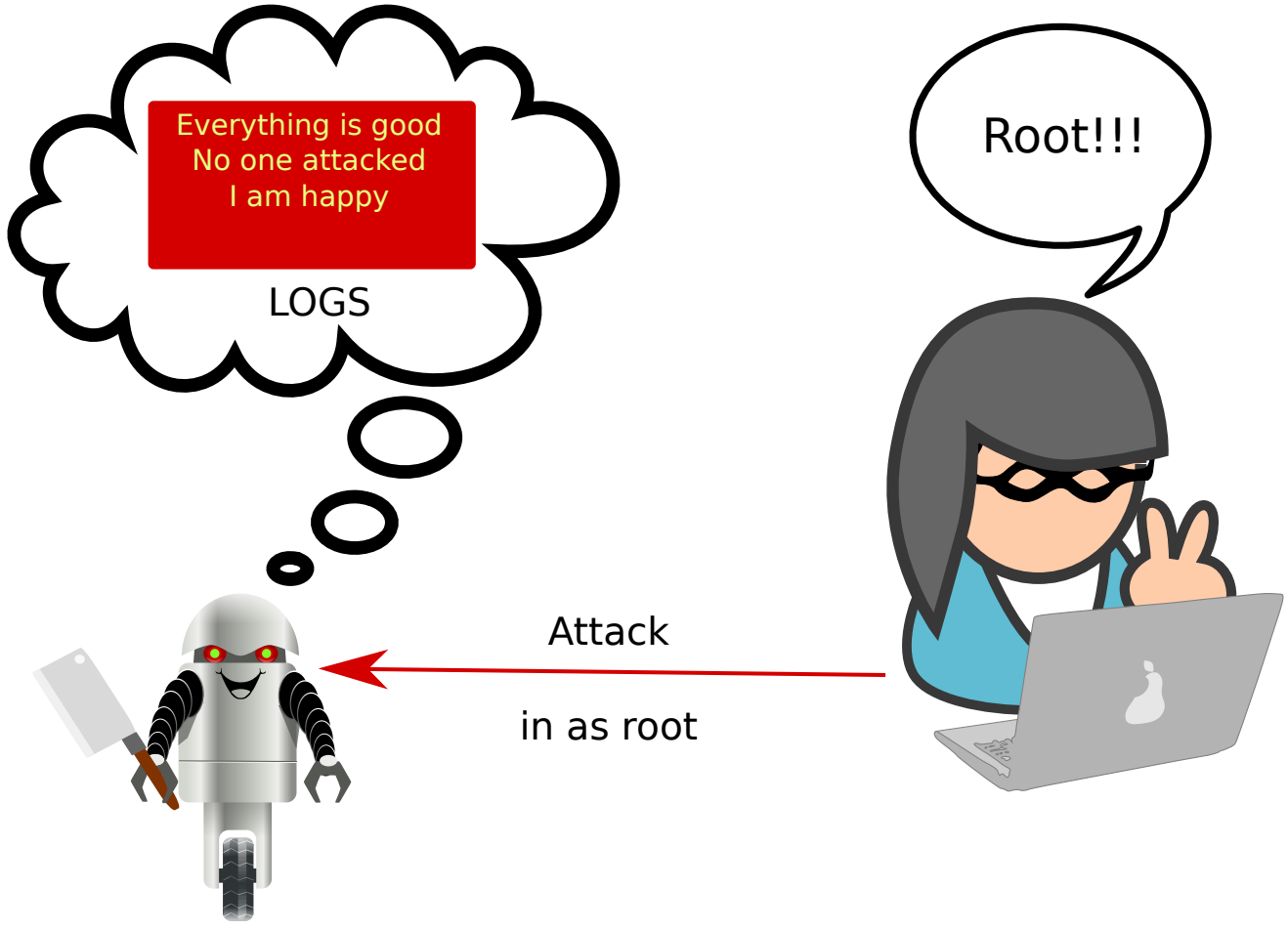
Fase 2: Atacante sin privilegios

- Funcionamiento normal
- Atacante tiene acceso, usuario sin privilegios



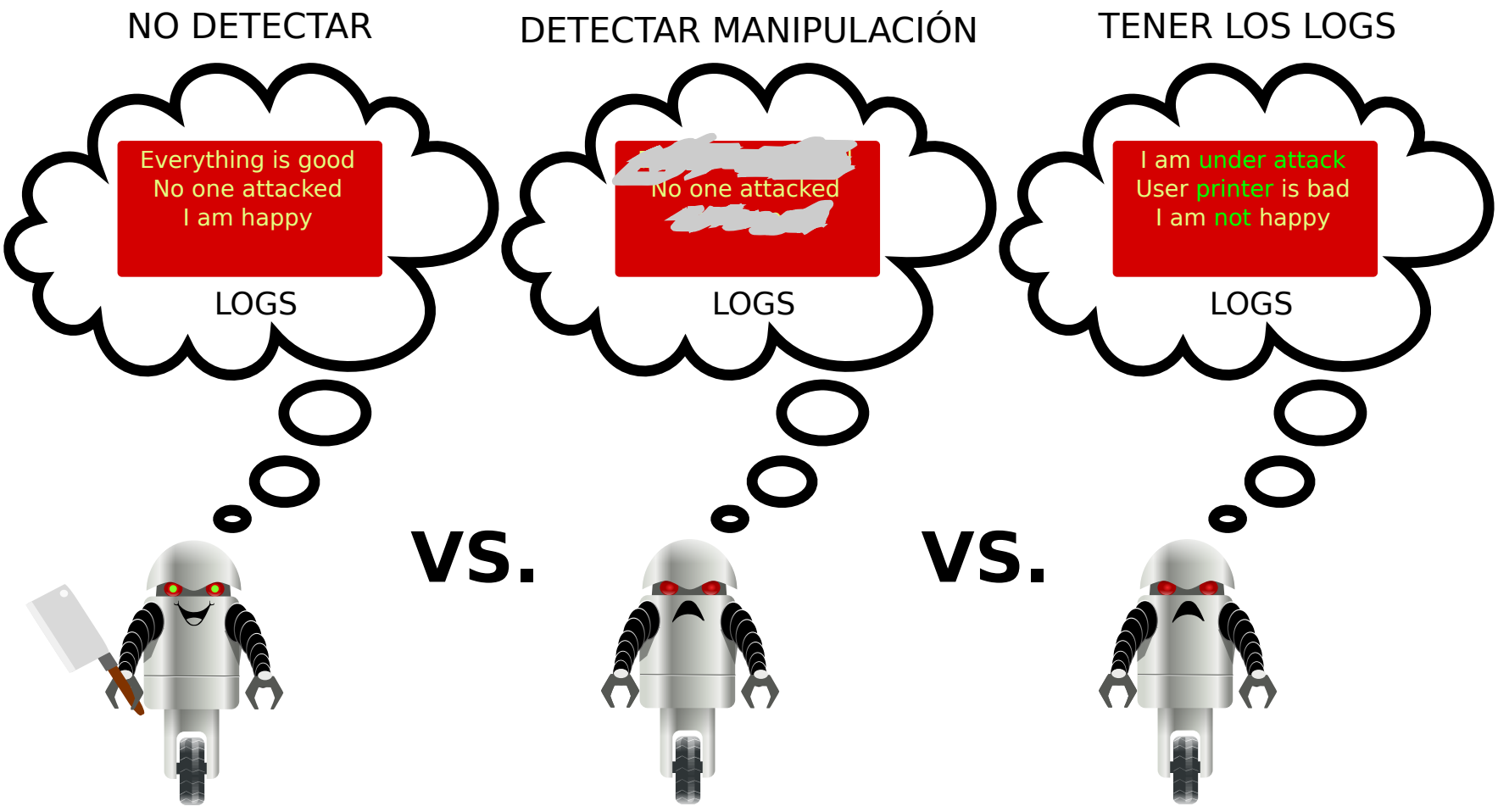
Fase 3: Atacante con privilegios

- El atacante se hace superusuario
- Borra sus trazas



Objetivo: detección

- Tras el ataque, que queden trazas en los logs



Estado del arte

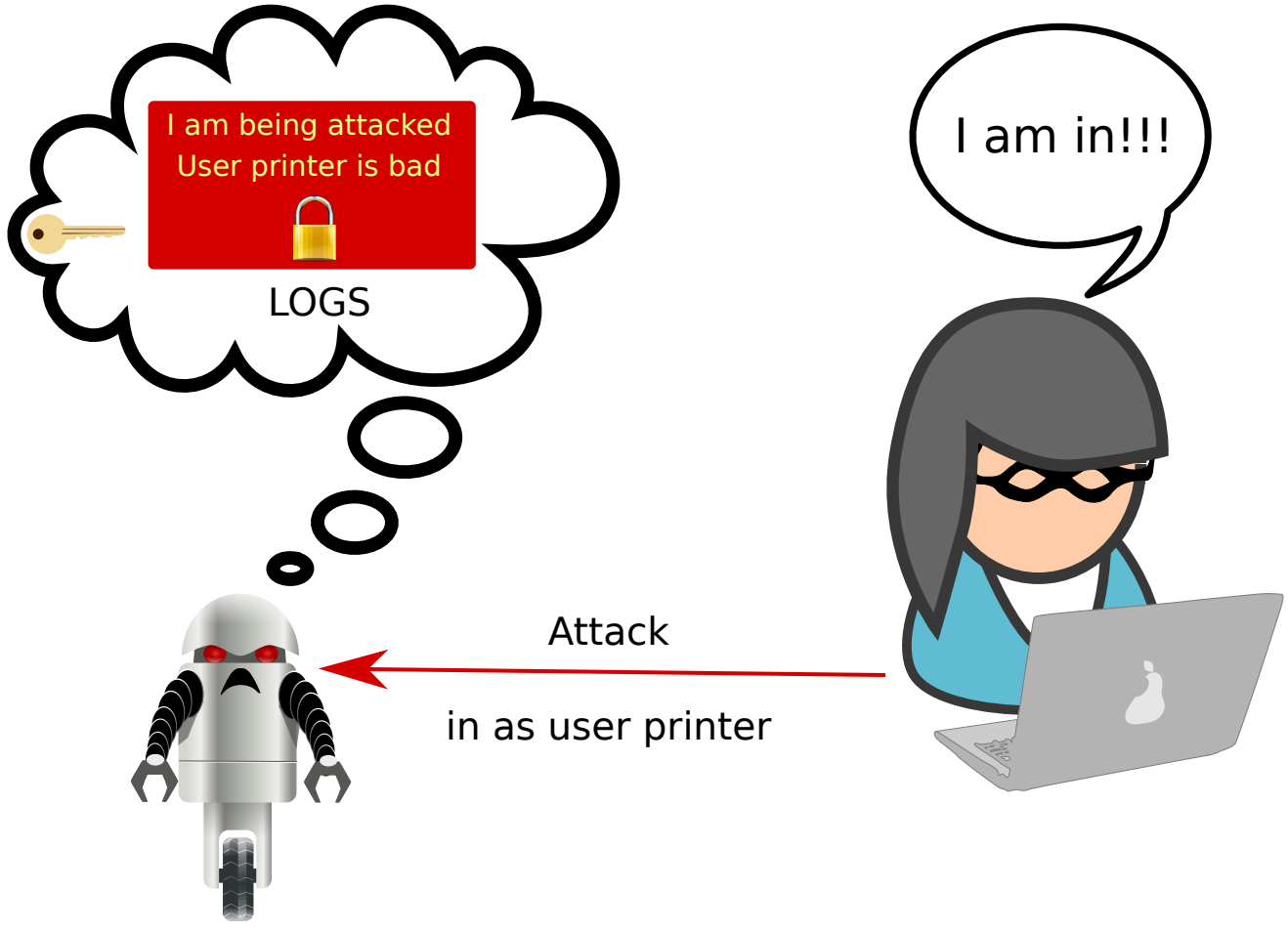
- Logs firmados
- Logs distribuidos
- Blockchain
- Ratchet
- OTP (one time pad), base de nuestra idea

Logs firmados

- Logs firmados (con una clave simétrica o asimétrica)
- Cuando el usuario es root, puede forjar logs

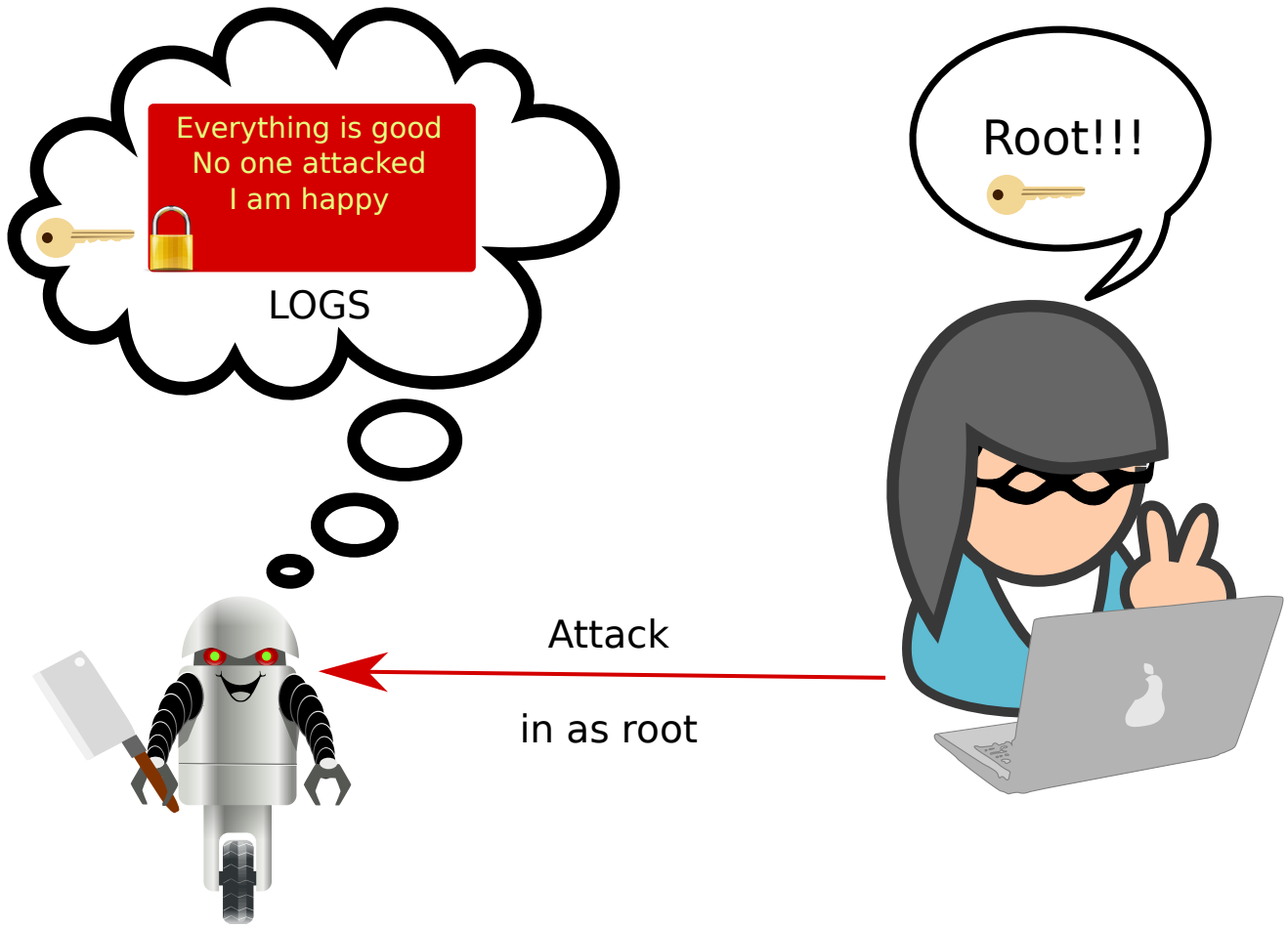
Fase 2: Atacante sin privilegios

- Funcionamiento normal
- Atacante tiene acceso, usuario sin privilegios
- Los logs capturan el ataque



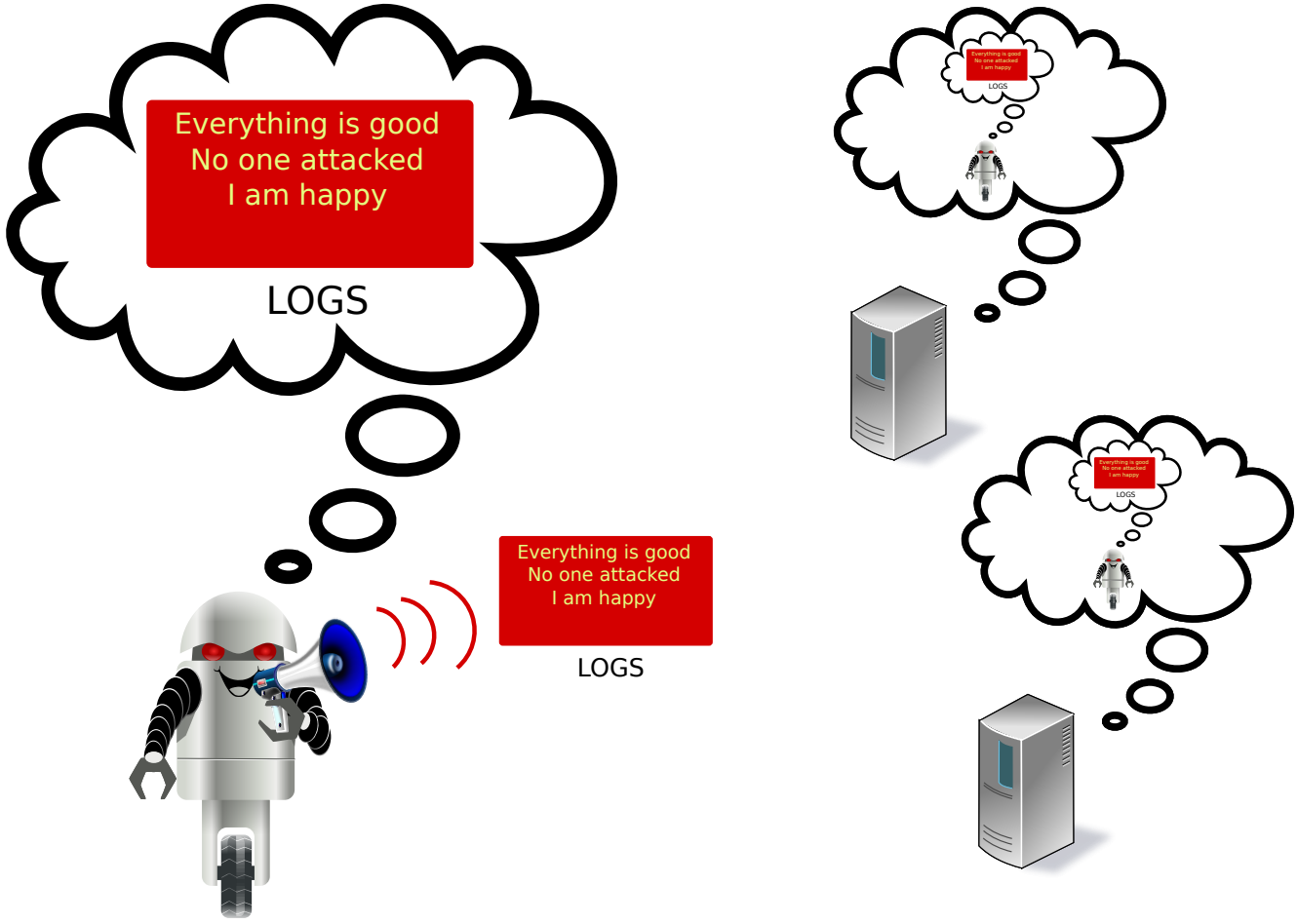
Fase 3: Atacante con privilegios

- El atacante se hace superusuario, inspecciona la memoria
- Forja logs con la clave privada y borra sus trazas
- Se puede evitar con hw. especial TrustZone, SIM, (no queremos)



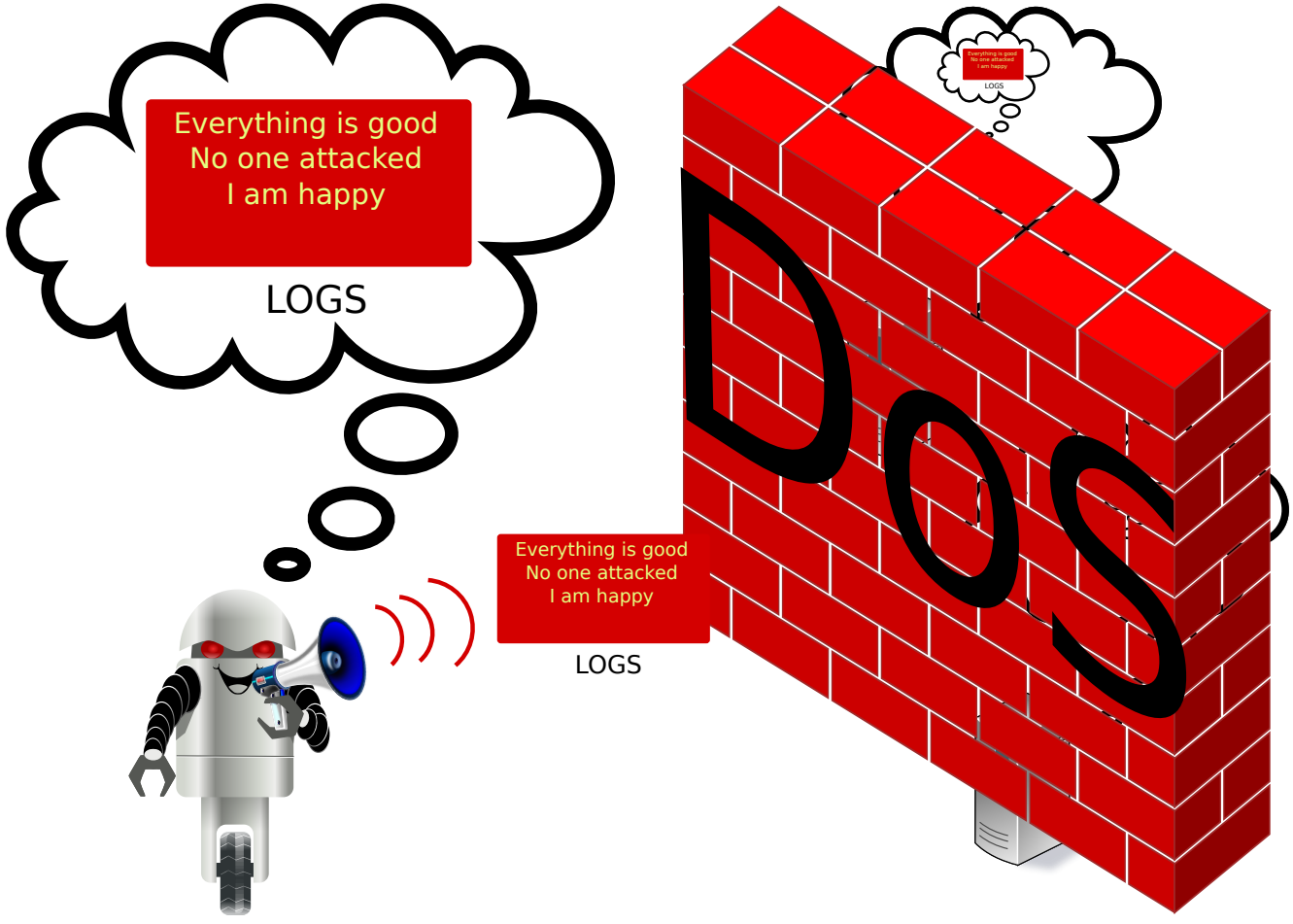
Logs distribuidos

- Radiar los logs a servidores que los guardan



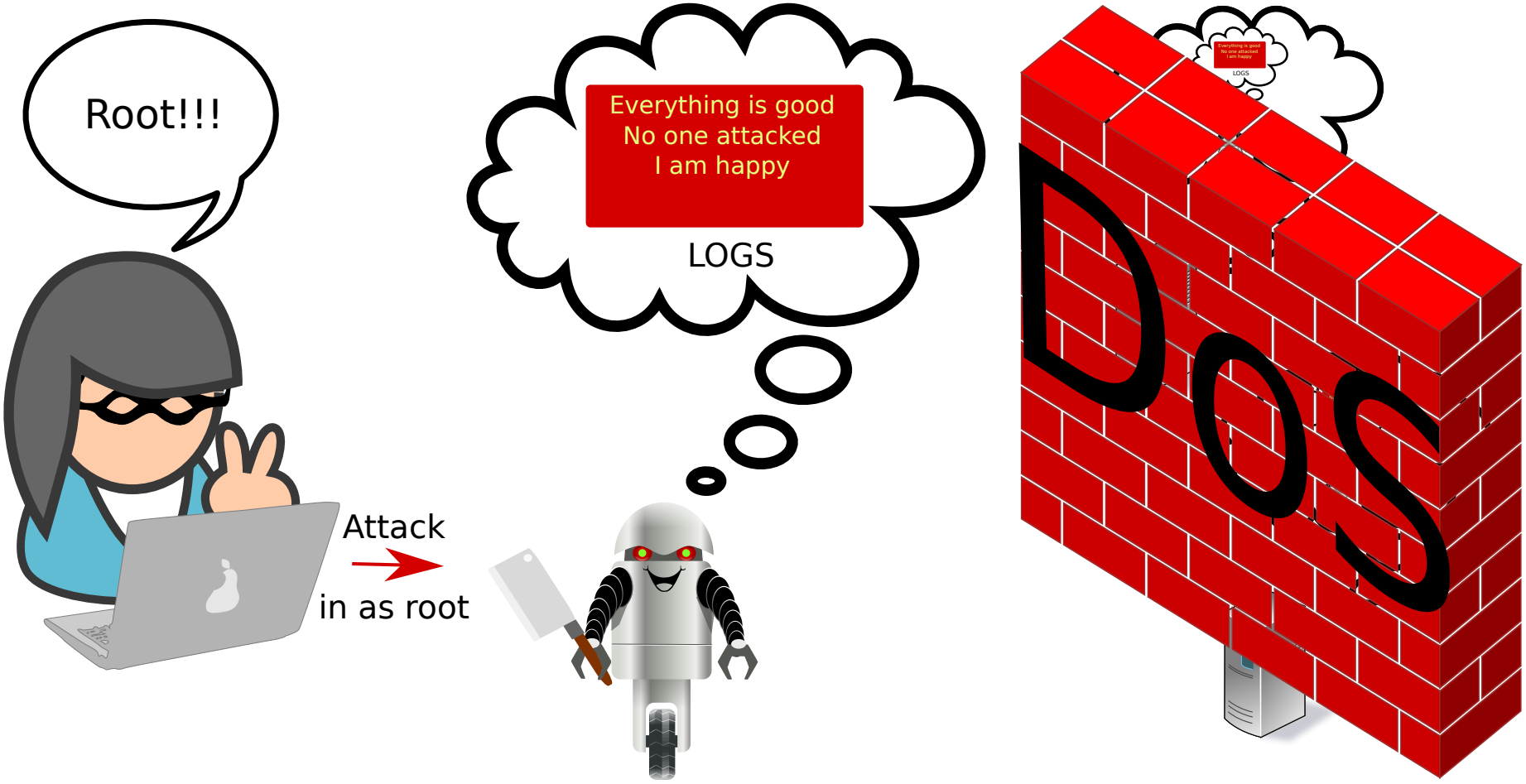
Logs distribuidos

- Más máquinas, más coste (administración y dinero)
- No vale para robots desconectados
- Perder la red (DoS) indistinguible de ataque



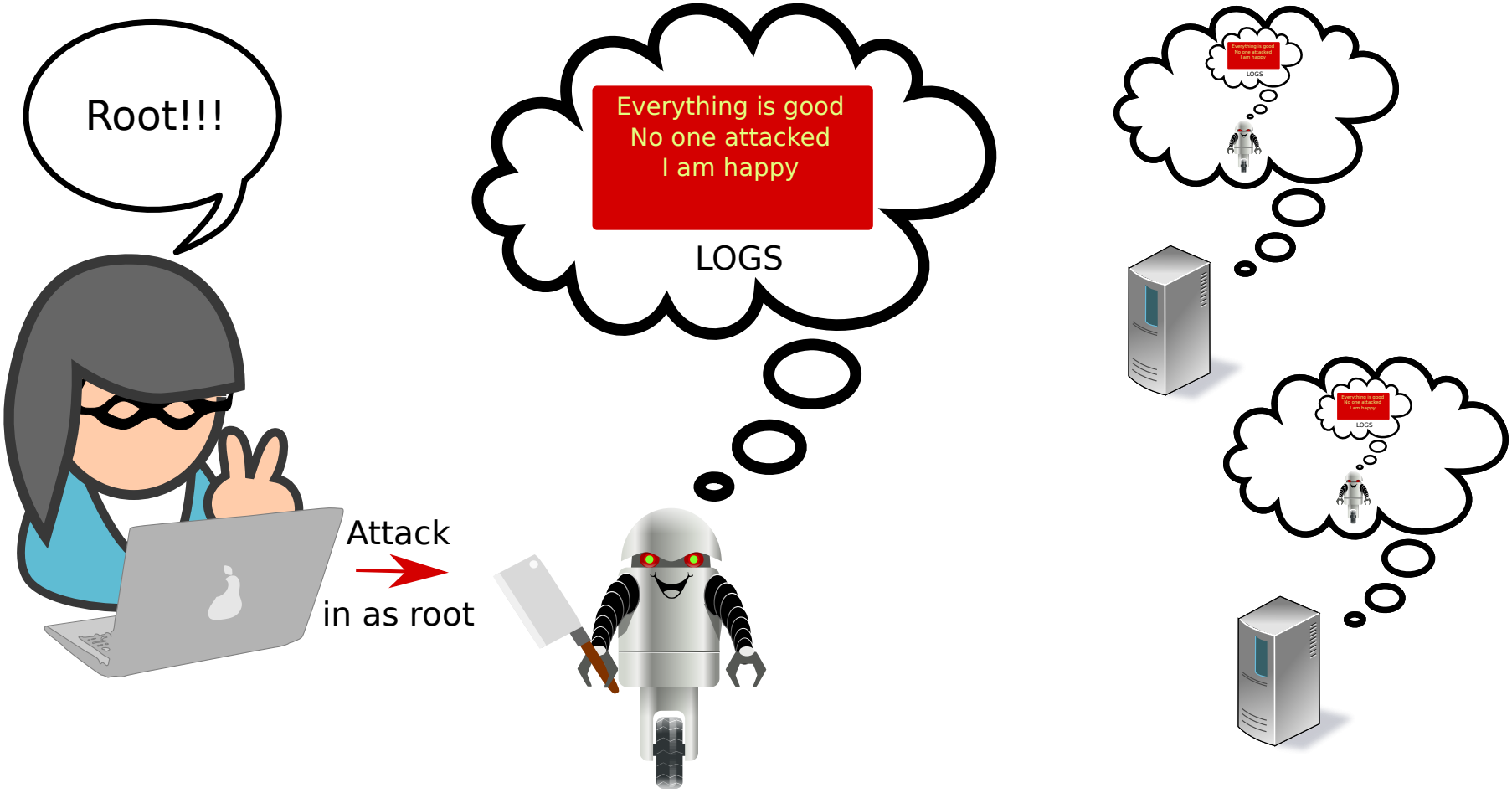
Logs distribuidos

- Atacas cuando no hay red



Logs distribuidos

- . . . y hecho

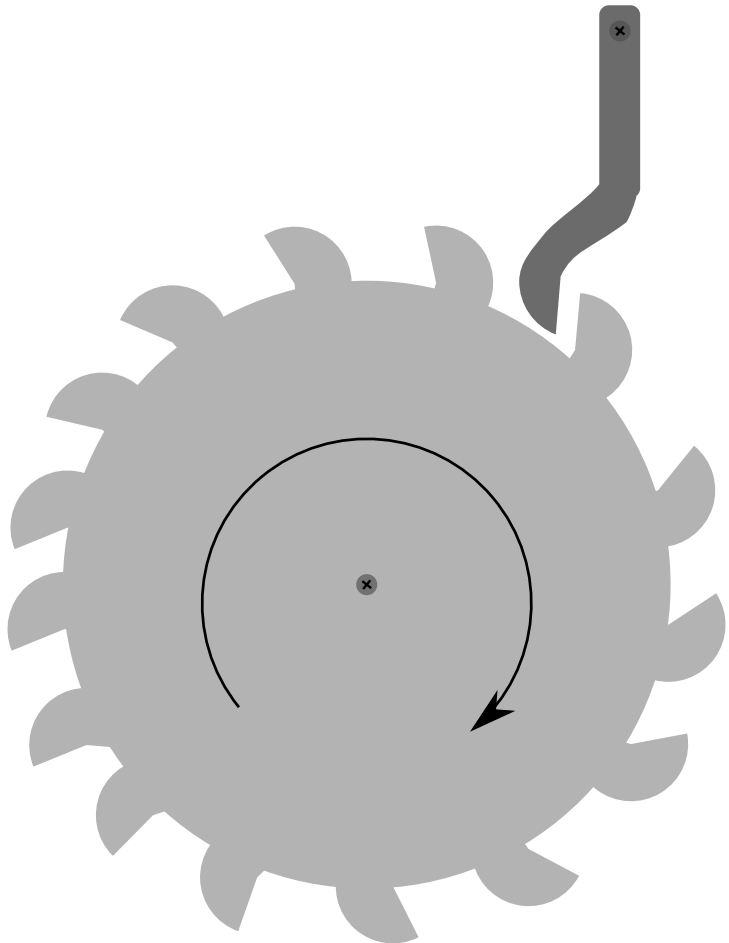


Blockchain

- Algo menos de administración pero. . .
- Mismas debilidades que los logs distribuidos
- . . . y además, extremadamente lento, consenso y prueba de trabajo para nada

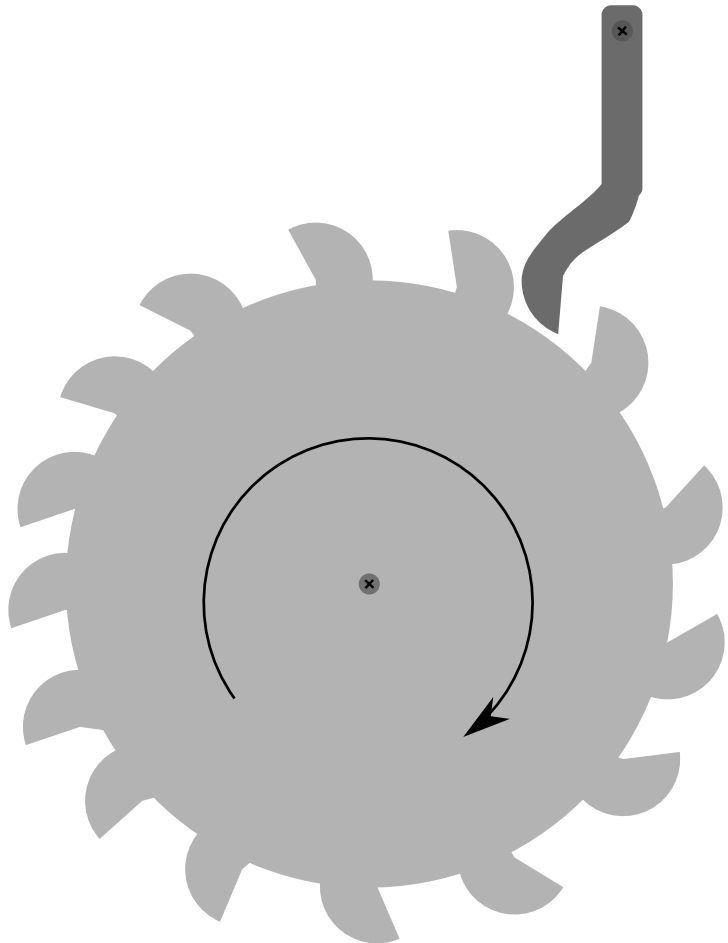
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



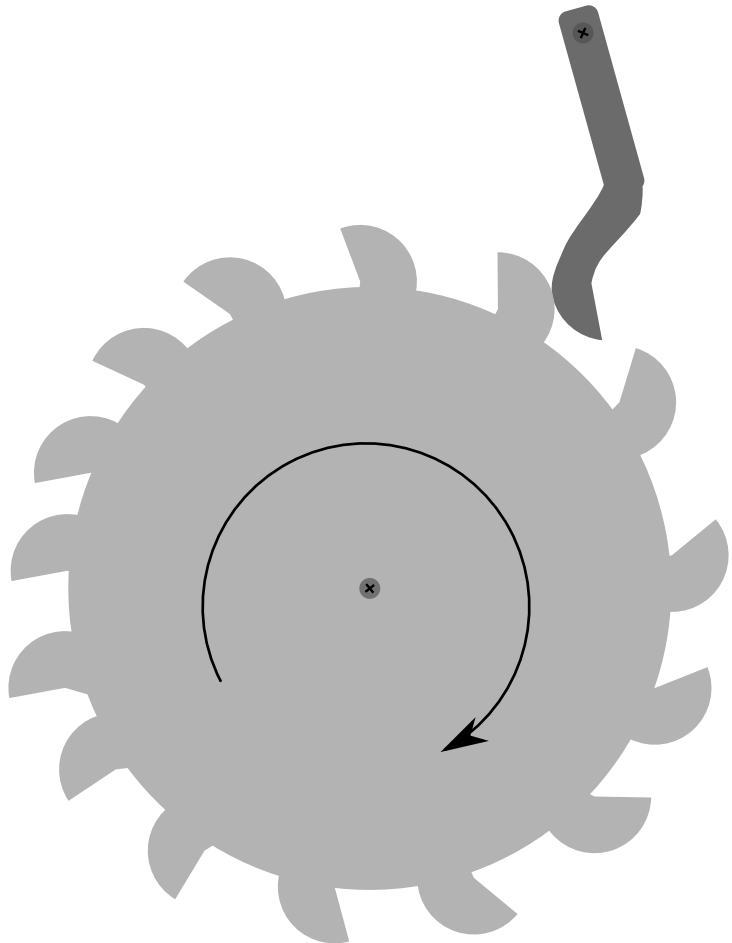
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



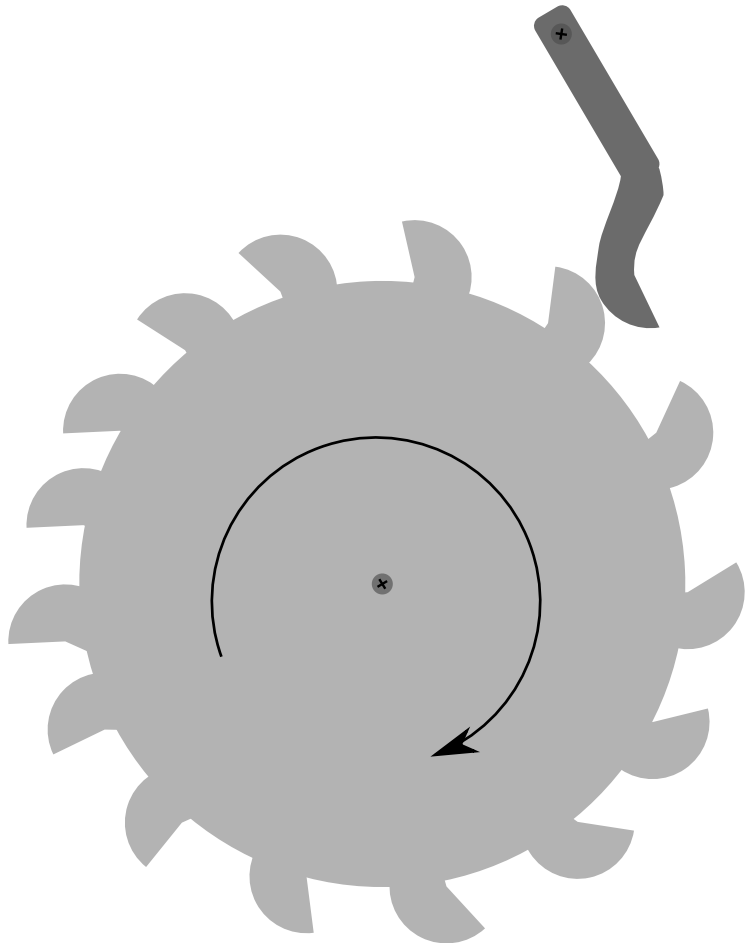
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



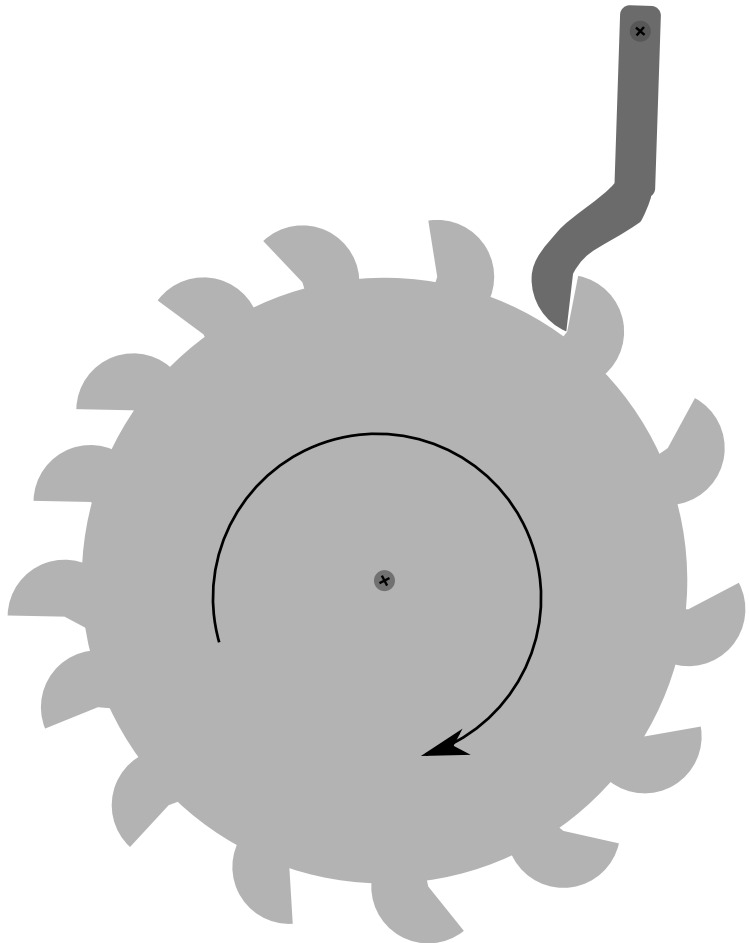
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



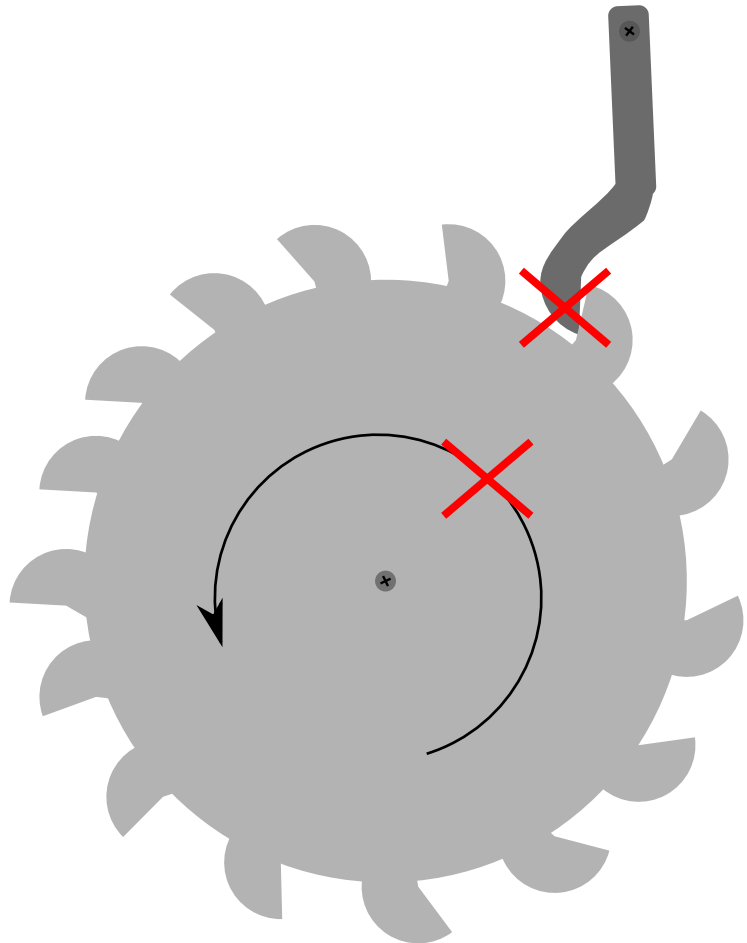
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



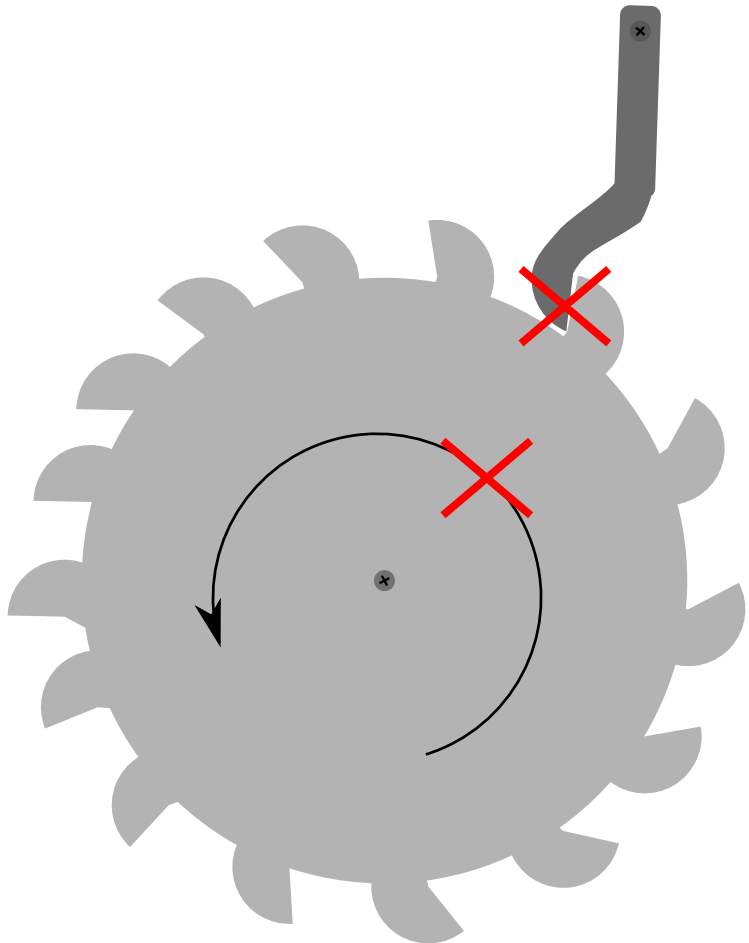
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



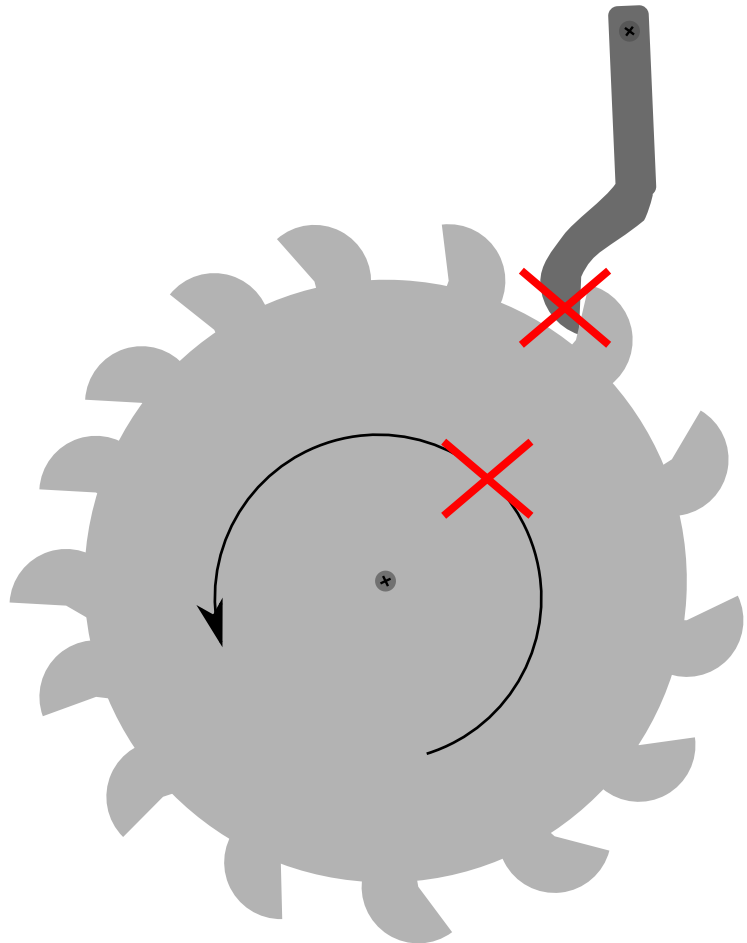
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



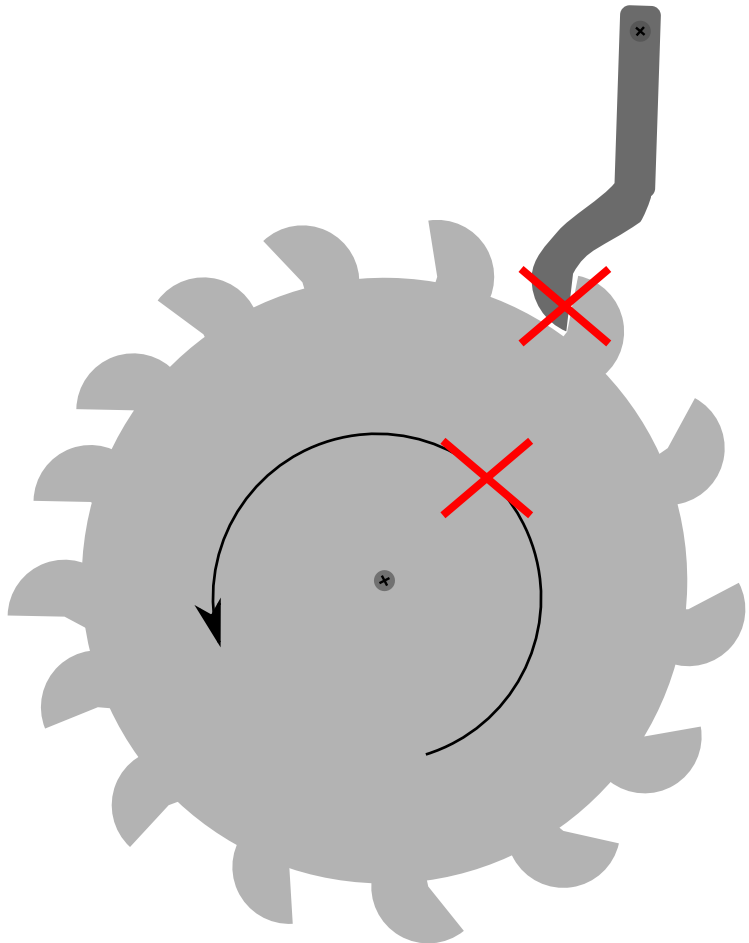
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



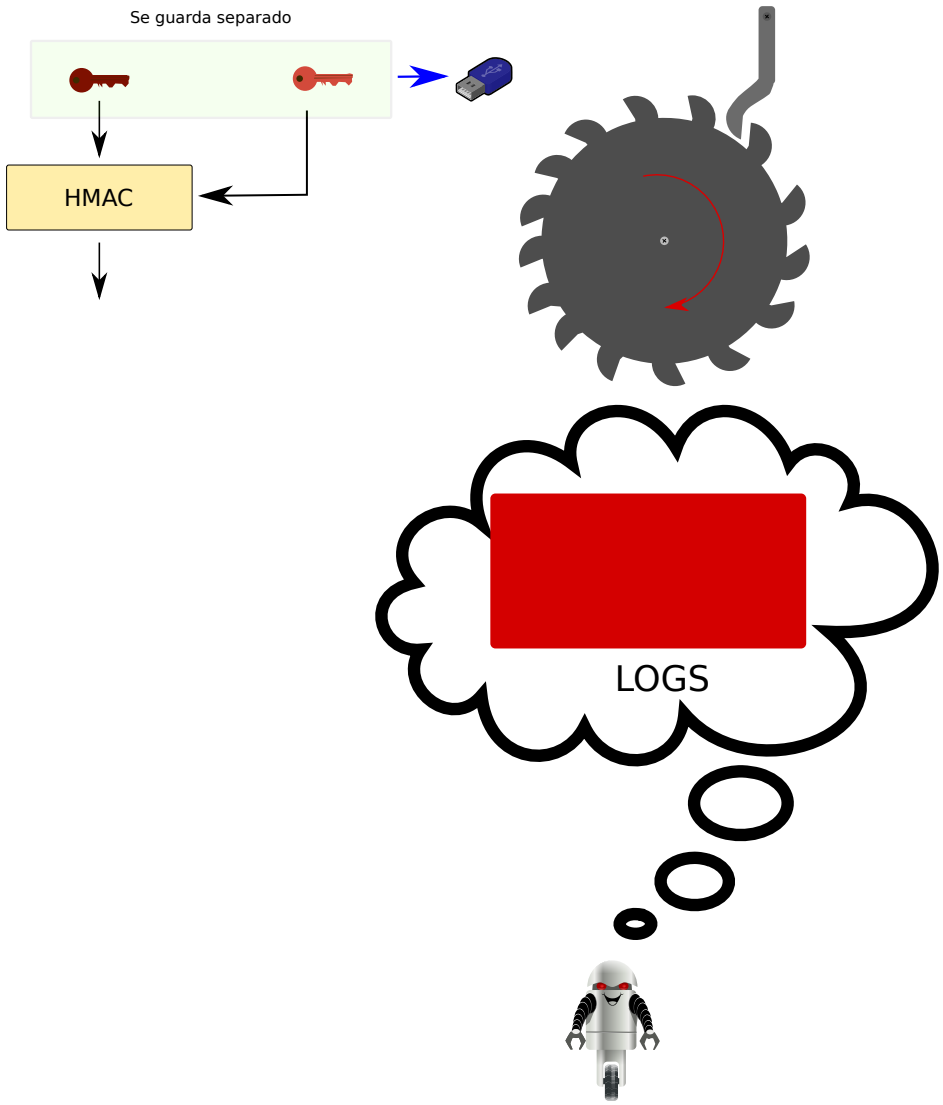
Ratchet

- Carraca
- Sólo avanza en un sentido
- *Merkle tree* lineal que va olvidando (similar a un CSPRNG)
- Claves iniciales se guardan en un sistema separado para comprobación



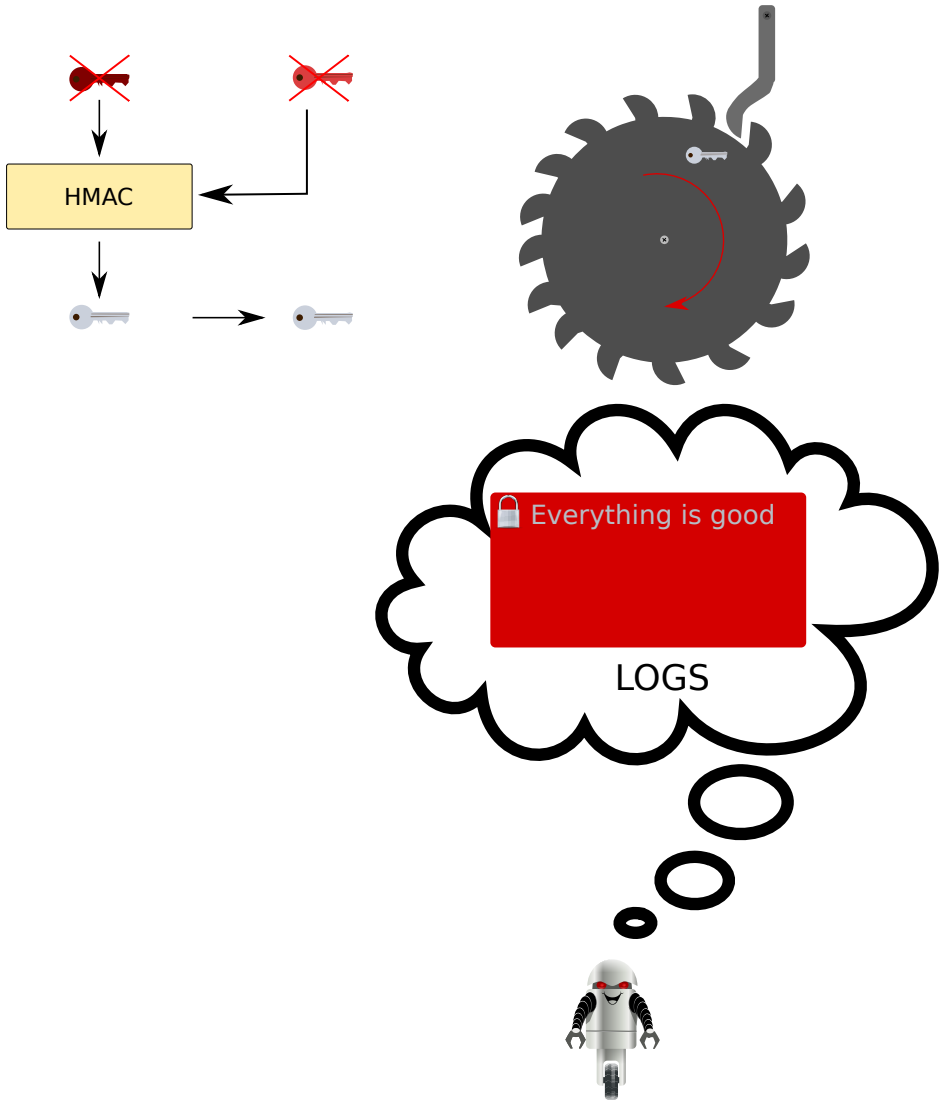
Ratchet

- Por ejemplo, crea nuevas claves usando una HMAC
- Poco estado inicial, pocos secretos, "se gastan": degradado lineal



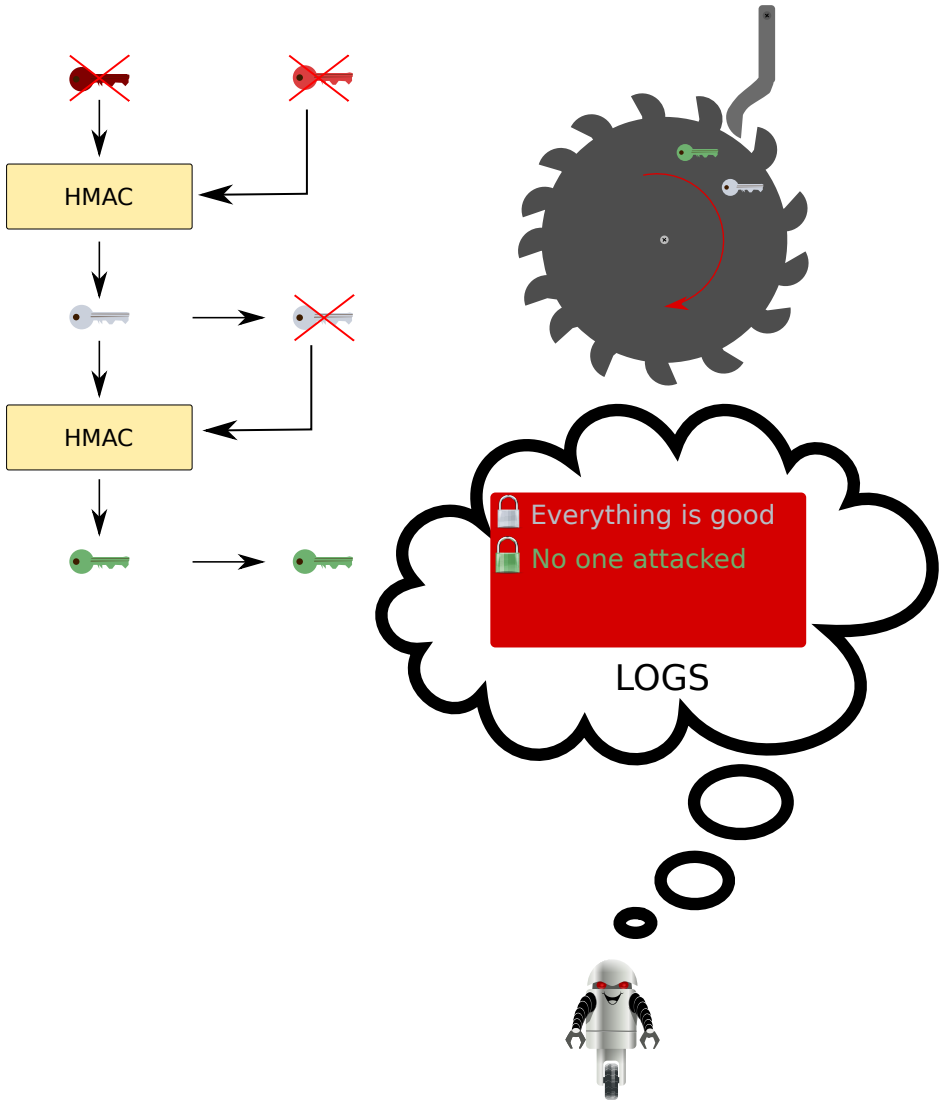
Ratchet

- Por ejemplo, crea nuevas claves usando una HMAC
- Poco estado inicial, pocos secretos, "se gastan": degradado lineal



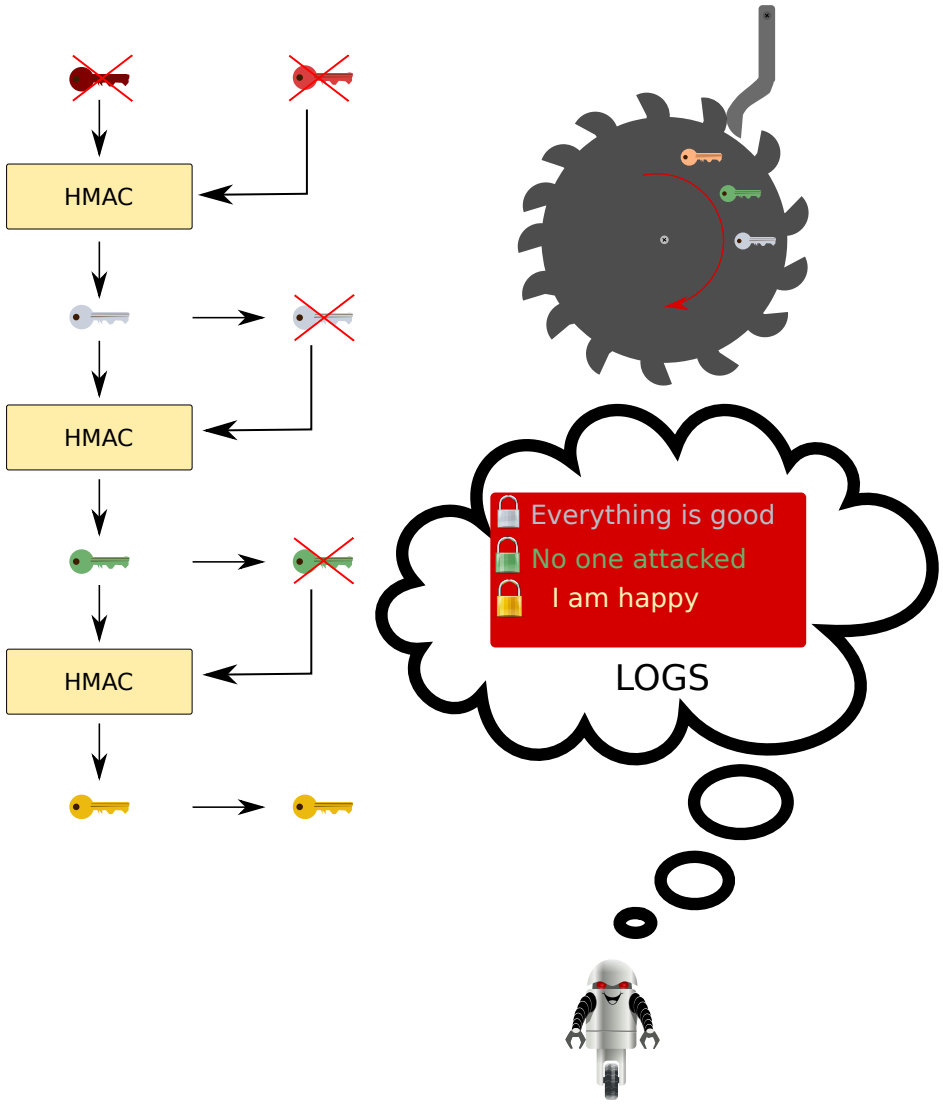
Ratchet

- Por ejemplo, crea nuevas claves usando una HMAC
- Poco estado inicial, pocos secretos, "se gastan": degradado lineal



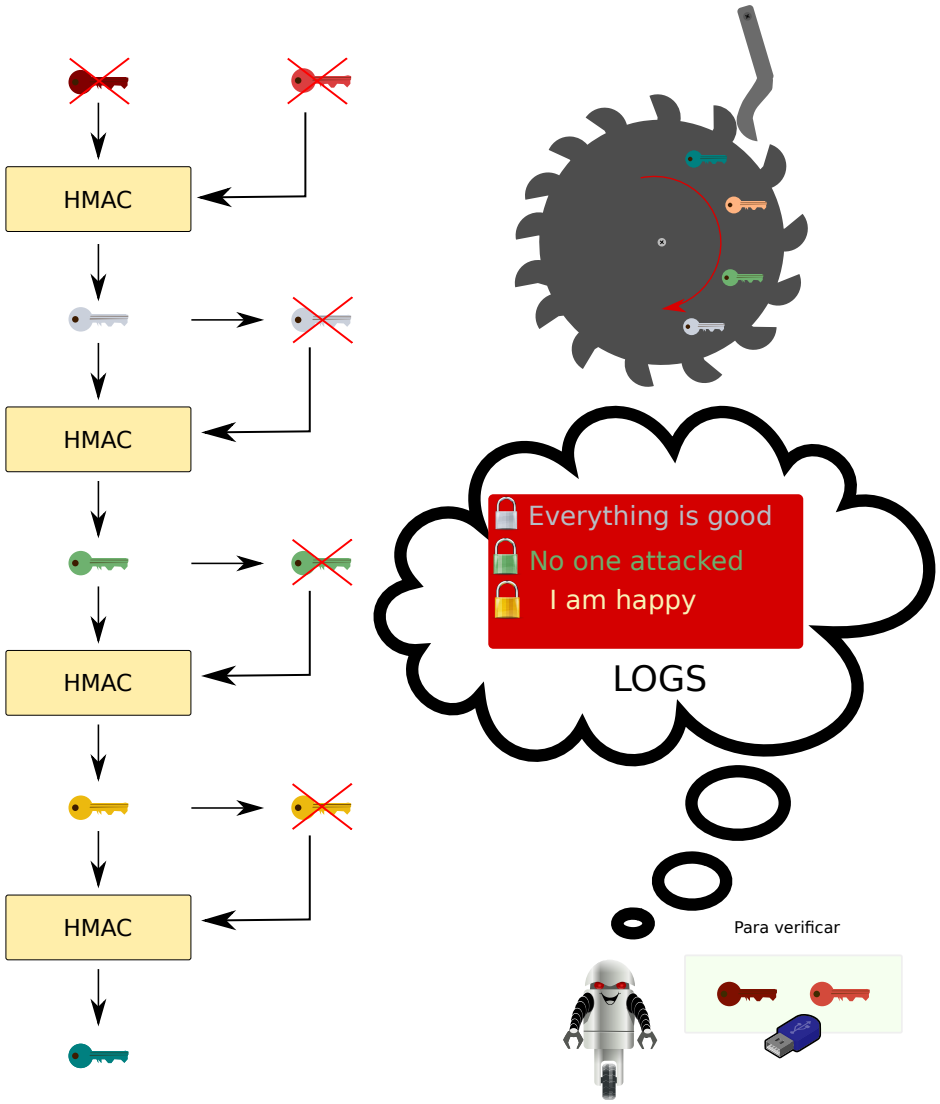
Ratchet

- Por ejemplo, crea nuevas claves usando una HMAC
- Poco estado inicial, pocos secretos, "se gastan": degradado lineal



Ratchet

- Claves guardadas y todo el procedimiento para comprobar
- Poco estado inicial, pocos secretos, "se gastan": degradado lineal



Nuestra idea

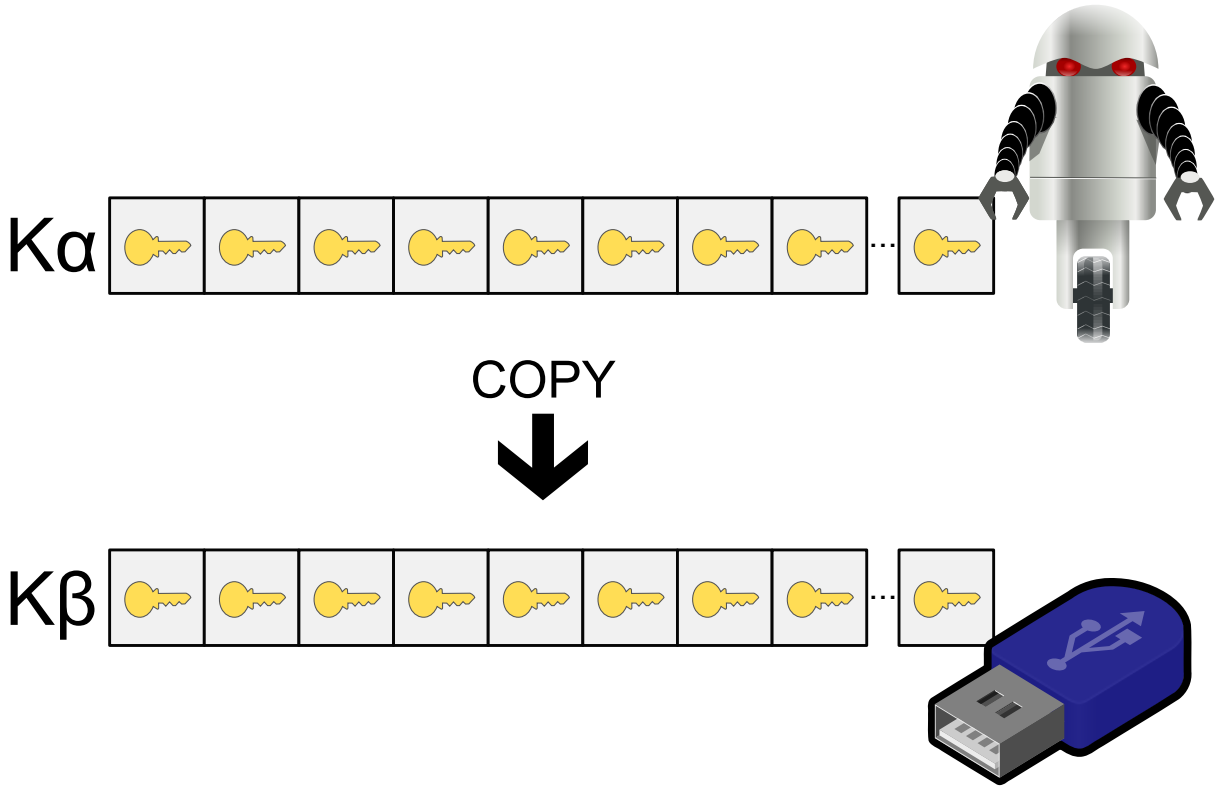
"USB 3.0 de 128 GB con hasta 150 MB/s de Velocidad de Lectura" **15.99 euros**

- ¿Por qué no aprovechamos esto?
- Lo podemos tratar como un token: *"algo que tienes"*

Nuestra idea

En **tiempo de configuración**:

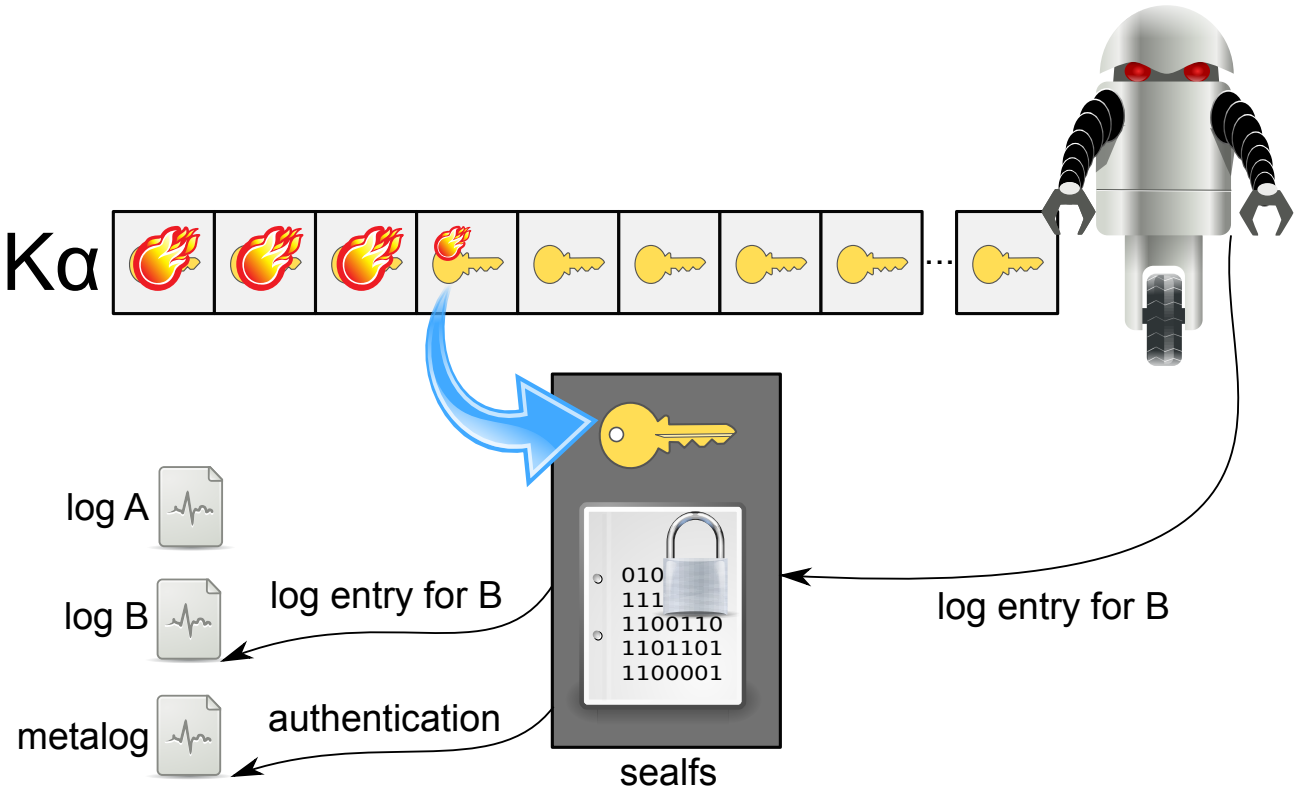
- Se crea un *keystream* largo: K_α
- Se almacena en la máquina (disco convencional)
- Se duplica en un dispositivo externo: K_β
- El dispositivo externo con K_β se desconecta y se guarda en un sitio seguro.



Nuestra idea

En **tiempo de ejecución** (con fases 1, 2 y 3 si el sistema es atacado):

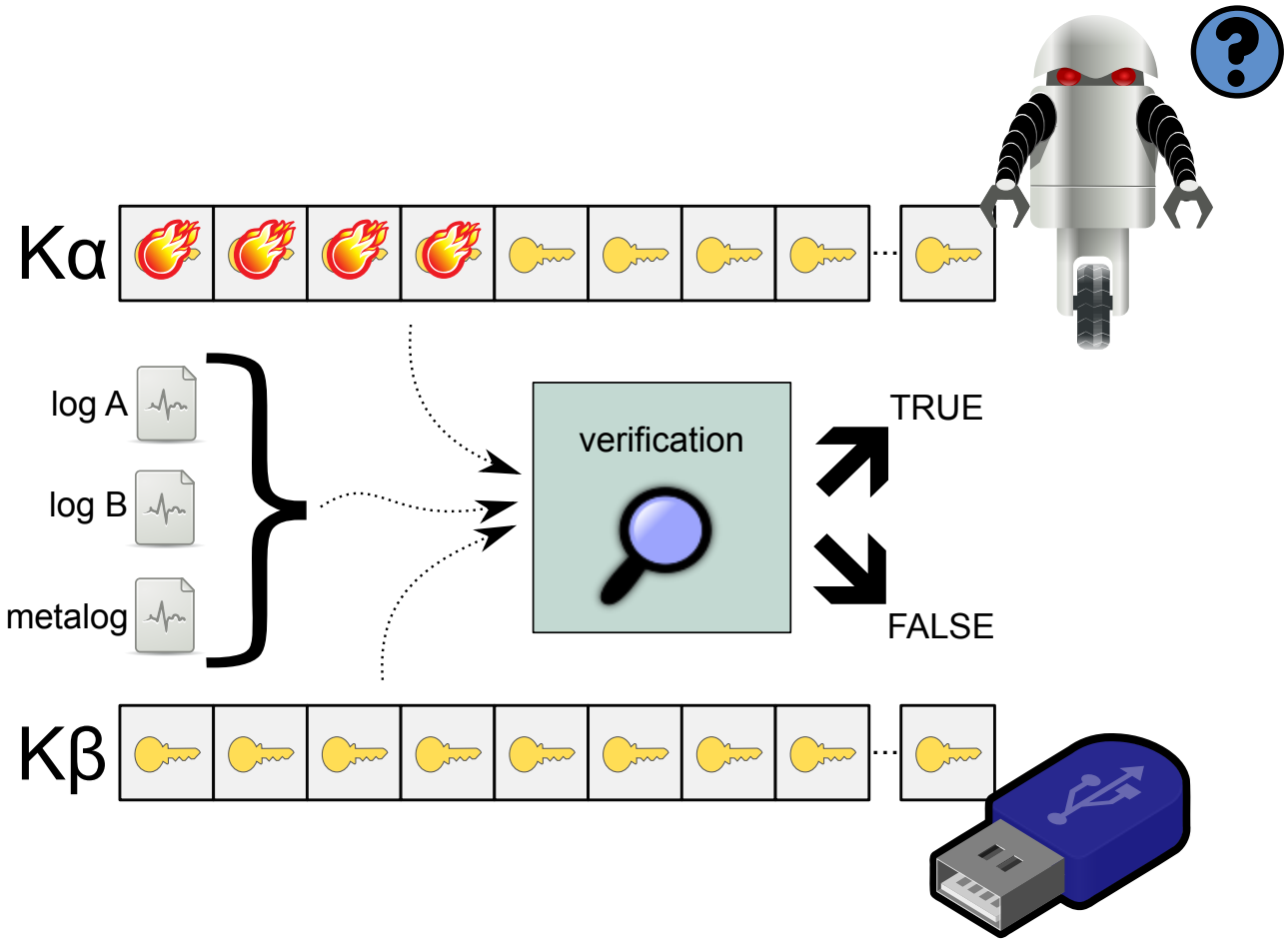
- Cada escritura en un log se autentica usando una porción de K_α
- Cada porción de K_α usada se **quema**
- Una vez quemada una parte de K_α, no se puede recuperar



Nuestra idea

En **tiempo de análisis forense**:

- Se usa dispositivo externo que contiene K_β para verificar las entradas de los logs.
- La verificación se ejecuta en otra máquina, que sea confiable.



Modelo de amenazas

Amenazas:

- El atacante puede ser local o remoto.
- Después de elevar privilegios, el atacante puede ejecutar código a cualquier nivel (de Ring 3 a -3).
- El atacante puede borrar o modificar cualquier dato después de la elevación.

Dependencias:

- Los dispositivos de almacenamiento son lo suficientemente grandes como para almacenar el *keystream* y un metalog.

Suposiciones:

- El sistema puede estar conectado o desconectado.
- La criptografía utilizada es segura.
- El atacante no puede desactivar o esquivar nuestro sistema de autenticación sin elevar privilegios antes.
- Una vez que un dato es borrado, no se puede recuperar.
- El auditor verifica los logs en otro sistema independiente y correcto.

Mitigaciones:

- El auditor puede comprobar si las entradas de los logs **que fueron generadas en las fases 1 y 2** han sido manipuladas o no.

One time pad

- Primera idea:
  - **one time pad**: hacer XOR de los datos escritos en los logs con las porciones de K_α
- No funciona:
  - Los logs se quedarían cifrados: la **confidencialidad** NO es el objetivo, el objetivo es la **autenticación y la integridad**.
  - Muchas partes de un log son predecibles (p. ej. horas/fechas) o conocidas (p. ej. entradas periódicas, prefijos, etc.). $\rightarrow$ esas partes permiten recuperar las porciones de K_α que han sido quemadas $\rightarrow$ permite falsear entradas.

Algoritmo: escritura en el log

1. Los datos (D_i) se escriben al final del fichero de log.
2. Se lee la porción C_i de K_α que toca usar y se sobrescribe.
3. Usando C_i como clave se calcula una HMAC H_i de los siguientes datos concatenados:
   - A ID que identifica el fichero de log (L).
   - B Offset en el fichero de log ($Loff_i$).
   - C Longitud de los datos escritos (Dsz_i).
   - D Offset en el fichero K_α para C_i ($Coff_i$).
   - E Los datos escritos en el log (D_i).
4. Se crea un registro con los datos A-D y la HMAC H_i generada:

$$R \leftarrow (L, Loff_i, Dsz_i, Coff_i, H_i)$$

5. Se escribe el registro R en el metalog.

IMPORTANTE: los pasos 2-5 deben ser **atómicos**.

Algoritmo: verificación de los logs

La verificación se basa en:

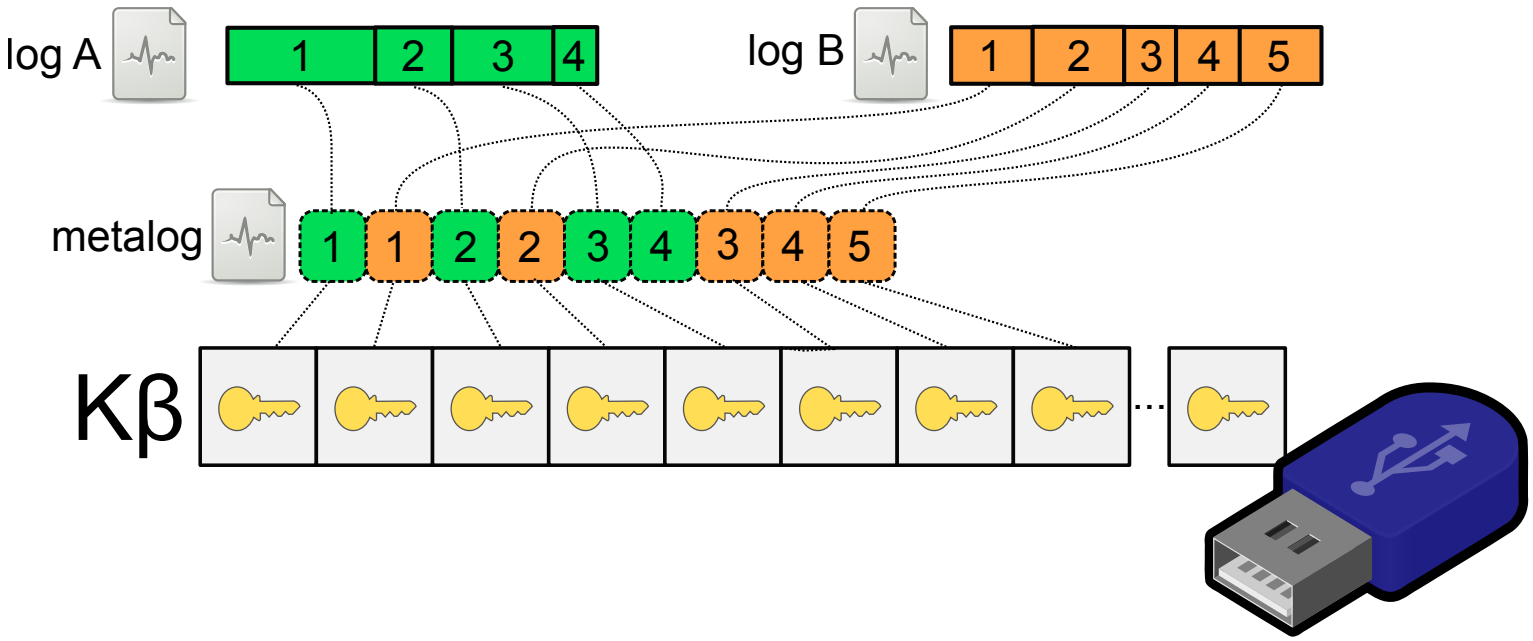
- K_α se quema secuencialmente y con K_β podemos saber hasta dónde se ha quemado.
- Cualquier modificación en los campos de un registro o los datos del log hace cambiar la HMAC.

Entonces, para verificar:

- Recorremos los registros del metalog comprobando que las HMACs coinciden y las áreas de los ficheros de log y de porciones de K_β son contiguas
- Las HMACs se generan obteniendo las porciones C_i de K_β (ya se quemaron en K_α).
- Los registros generados tienen que cubrir el área quemada de K_α.

Algoritmo: verificación de los logs

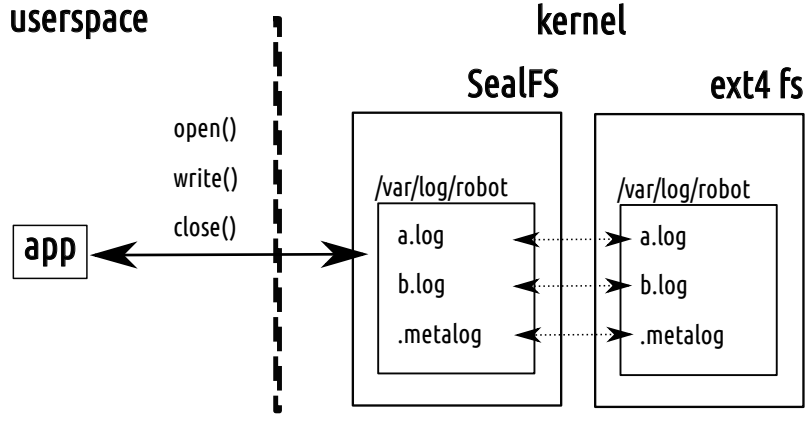
- Si la HMAC es segura el atacante no puede forjar entradas falsas del metalog para las que no tiene C_i (i.e. zonas quemadas).
- Dada la contigüidad, el atacante no puede eliminar registros del metalog, truncar (por ningún extremo) los ficheros de log o eliminar entradas de los ficheros de log:
  - Dados dos registros contiguos en el metalog, sus porciones correspondientes de K_β deben ser contiguas.
  - Los registros que pertenecen a un mismo fichero de log L están ordenados en el metalog $\rightarrow$ las áreas definidas por esos registros deben ser contiguas.



Implementación: sealfs

- El esquema es sencillo, la implementación no tanto.
- Implementación en un módulo de kernel para Linux basado en `wrapfs`.
- Sistema de ficheros *apilable* $\rightarrow$ ofrece los mismos ficheros del directorio subyacente (ext4).

```
$>  logdir=/var/log/robot
$>  stat -f -c '%T' $logdir
ext2/ext3
$>  ls -l $logdir
total 0
-rw-r--r-- 1 root root 0 Apr 10 14:21 a.log
-rw-r--r-- 1 root root 0 Apr 10 14:21 b.log
$>  mount -t sealfs   $logdir $logdir -o kpath=/kalpha
$>  ls -l  $logdir
total 0
-rw-r--r-- 1 root root 0 Apr 10 14:21 a.log
-rw-r--r-- 1 root root 0 Apr 10 14:21 b.log
$>  mount | fgrep  $logdir
/var/log/robot on /var/log/robot type sealfs (rw)
```



Implementación: sealfs

Ventajas:

- Autentica las escrituras en los logs de forma transparente.
- No hay que modificar las aplicaciones: escriben los ficheros como antes usando las llamadas al sistema $\rightarrow$ compatible hacia atrás.
- Fuerza la apertura **append-only** de los ficheros.
- Prohíbe operaciones peligrosas (p. ej. `mmap`).

Implementación: sealfs

- No es sencillo conseguir el *offset* en el que se escribe en el fichero subyacente.
- Concurrencia:
  - Mutex por fichero para obtener/actualizar el offset.
  - Mutex global para serializar los registros del metalog.
- Los ficheros se identifican con su **i-nodo** $\rightarrow$ rotación de logs con un simple `mv`.
- `sealfsd`: demonio para **requemar** K_α y sincronizarlo con el dispositivo de almacenamiento en background.

Rendimiento

- Medido en una máquina limitada (Intel Xeon, 4GB RAM).
- Verificación muy rápida (1500 registros en 0.01 s).
- Es suficientemente rápido para escribir logs: aprox. 0.02 ms para escribir una entrada de log de 100 caracteres. No suficientemente rápido para entrada/salida intensiva (no es el objetivo).
- Hay una opción de montaje para hacer escrituras síncronas: es demasiado caro.
- HMAC-SHA1 $\rightarrow$ C_i de 20 bytes $\rightarrow$ un pendrive de 32GB soporta hasta $1.6e9$ escrituras. Cuesta \$5.

Conclusiones

- Resumen:
  - Esquema simple.
  - Operación local y autónoma.
  - Sin necesidad de hardware especializado.
  - Compatibilidad hacia atrás y despliegue inmediato.
  - Sin contraseñas: *algo que tienes*.
  - Viable para logs, verificación rápida.
- Trabajo futuro:
  - Esquema híbrido sealfs/ratchet.
  - Optimizar implementación (mecanismos, concurrencia, etc.).
- Git:

  `https://gitlab.etsit.urjc.es/esoriano/sealfs`
- Paper (preprint):

  `https://gsyc.urjc.es/esoriano/sealfs-preprint.pdf`