A file system for safely interacting with untrusted USB flash drives

Ke Zhong, Zhihao Jiang[★], Ke Ma[★], and Sebastian Angel
University of Pennsylvania [★]Shanghai Jiao Tong University

Most Storage has moved to cloud!

USB flash drives remain popular

- **Legacy data**
- **No network connections**
- **Store confidential data**
  - Bitcoin keys
  - Medical records
  - ID photos

USB stack has several issues

- **Trust-by-default design principle**

- **Devices can bypass kernel and access memory (DMA)**

- **Driver code tends to be buggy**
  - There are many drivers by third party producers

- **Masquerade as other devices**
  - A device could declare to be a keyboard

USB stack has several issues

- **Trust-by-default design principle**
- **Devices can bypass kernel and access memory (DMA)**
- **Driver code tends to be buggy**
  - There are many drivers by third party producers
- **Masquerade as other devices**
  - A device could declare to be a keyboard

Could be exploited by a malicious flash drive

USB stack has several issues

- **Trust-by-default design principle**
- **Devices can bypass kernel and access memory (DMA)**
- **Driver code tends to be buggy**
  - There are many drivers by third party producers
- **Masquerade as other devices**
  - A device could declare to be a keyboard

Previous work

- **Packet filtering**
  - Cinch: Security’16
  - USBFilter: Security’16

- **Device authentication**
  - ProvUSB: CCS’16

- **Sandbox the device**
  - GoodUSB: ACSAS’15

Limitation

- **Packet filtering**
  - Malicious payload that changes dynamically avoids rule-based detection

- **Device authentication**
  - Require new hardware/kernel modifications

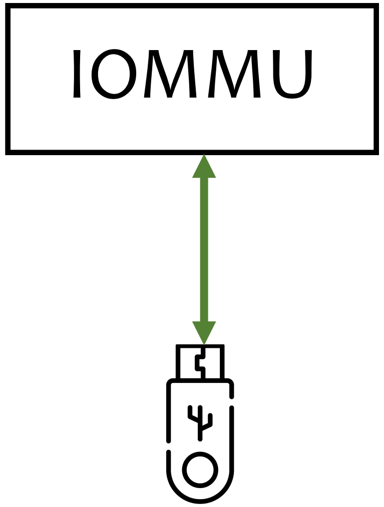
- **Sandbox the device**
  - False negative (i.e., a device is malicious but sandbox says it's ok)

We propose ***RBFuse*,** which is a file system that accesses flash drives without interacting with the USB stack on the host machine

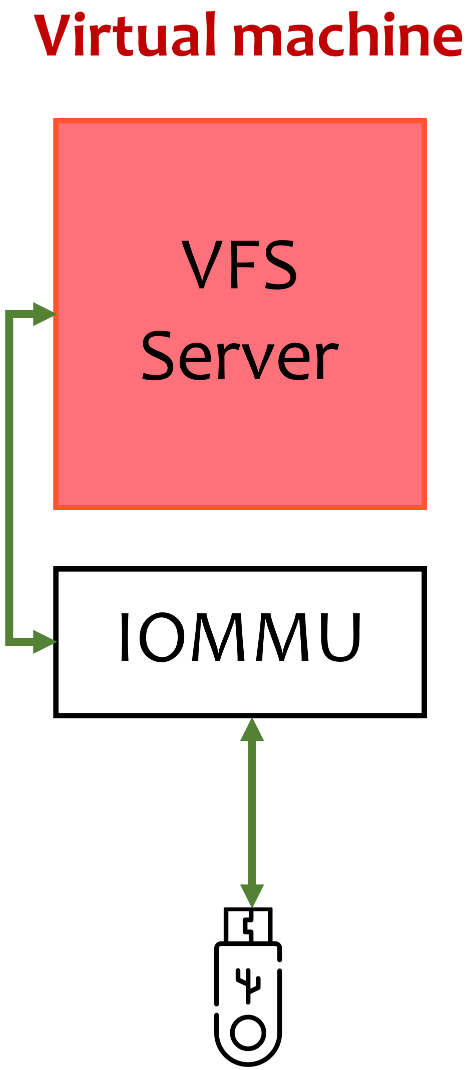
Key idea

RBFuse remaps memory space of
host controller to a virtual machine,
and exports file system of flash drives as
a mountable virtual file system

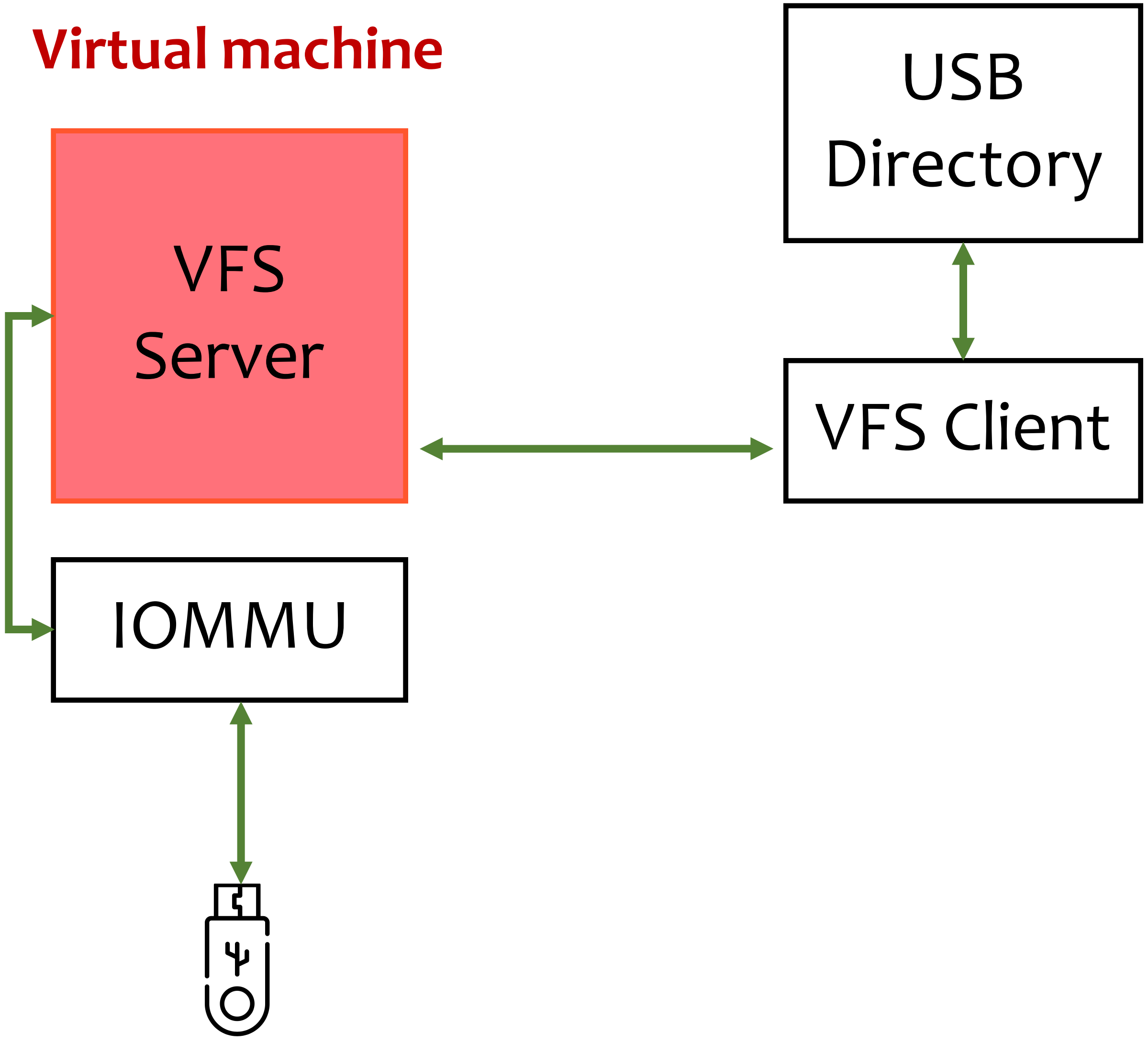
System overview



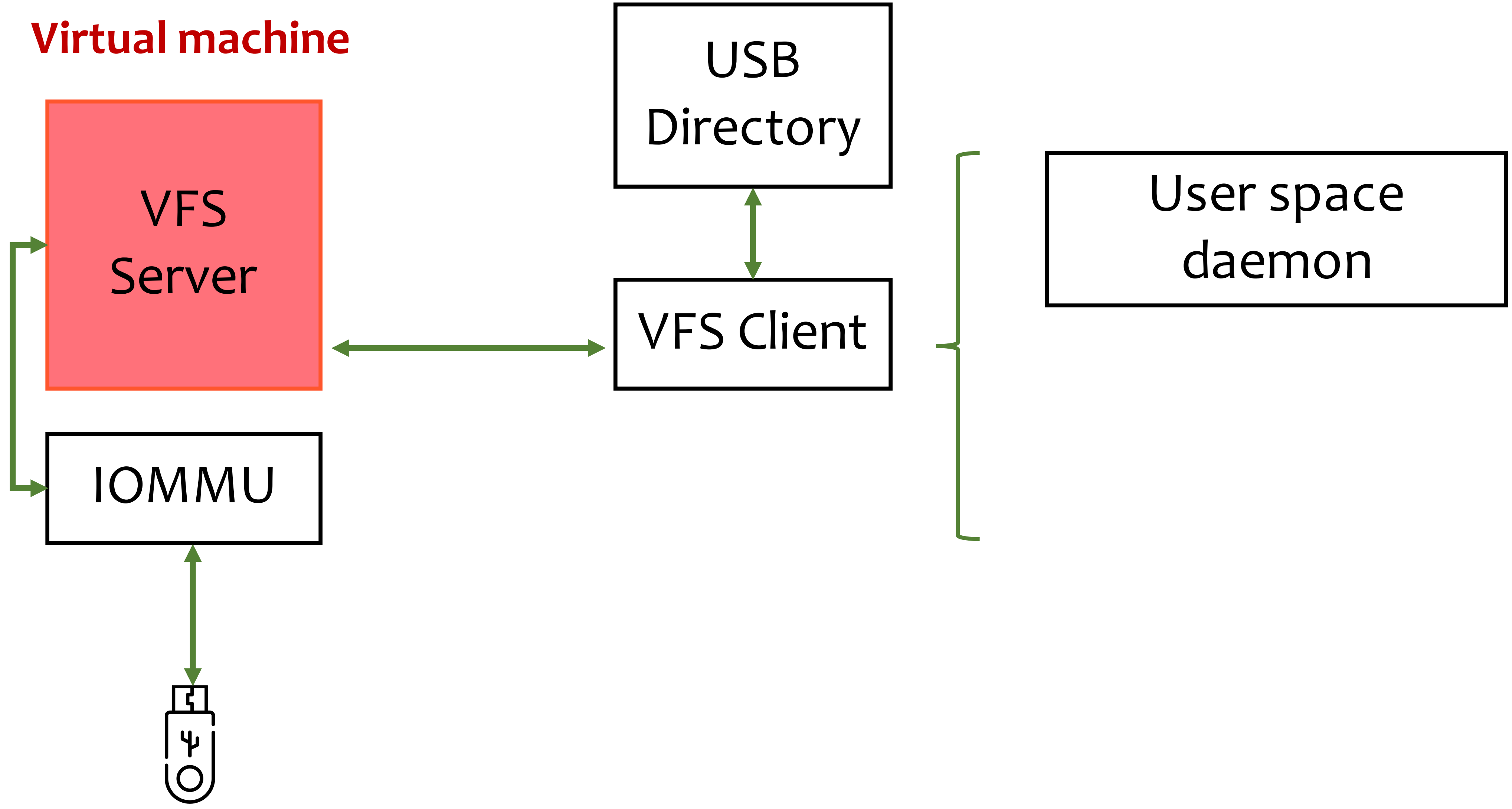
System overview



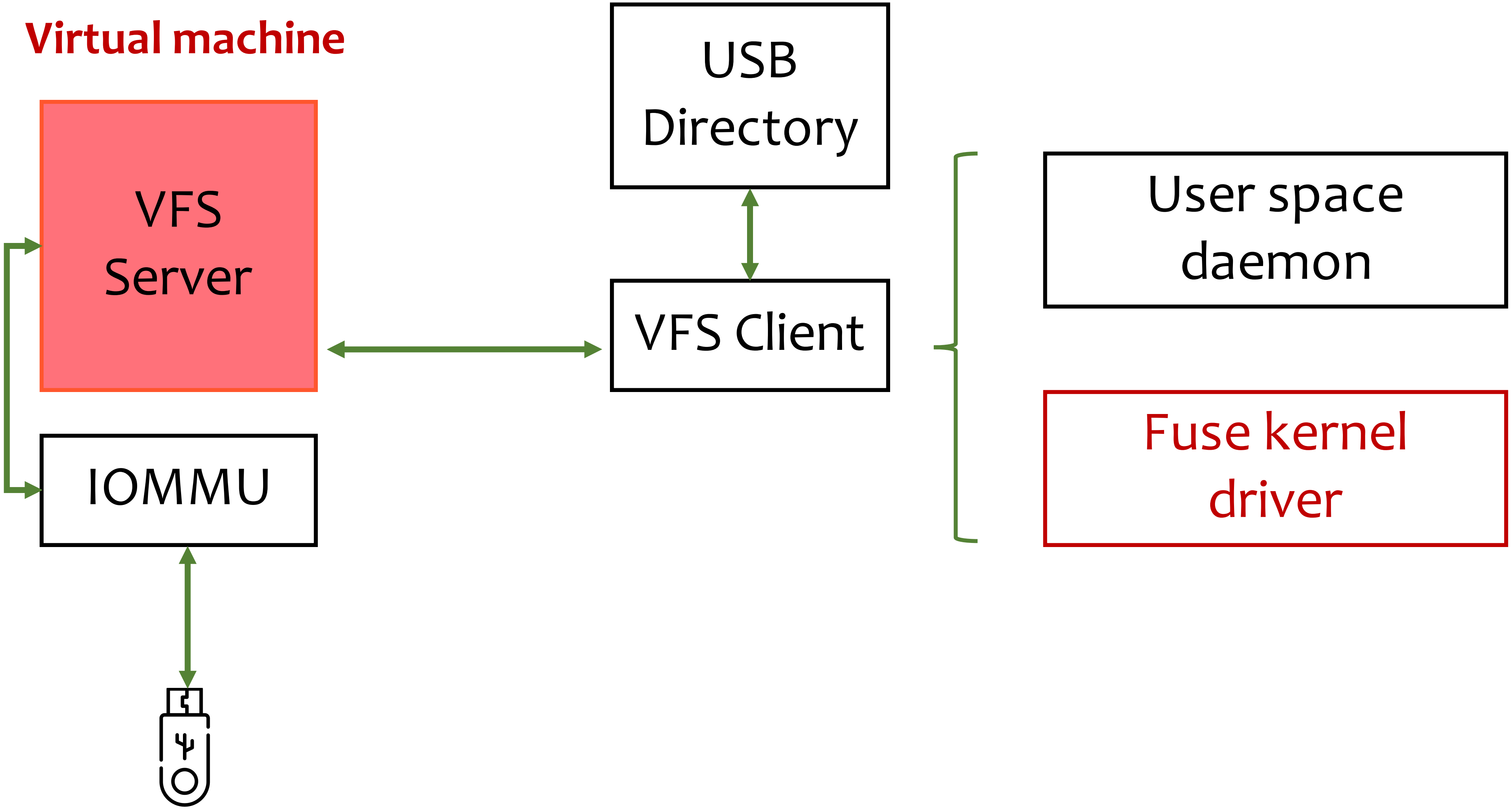
System overview



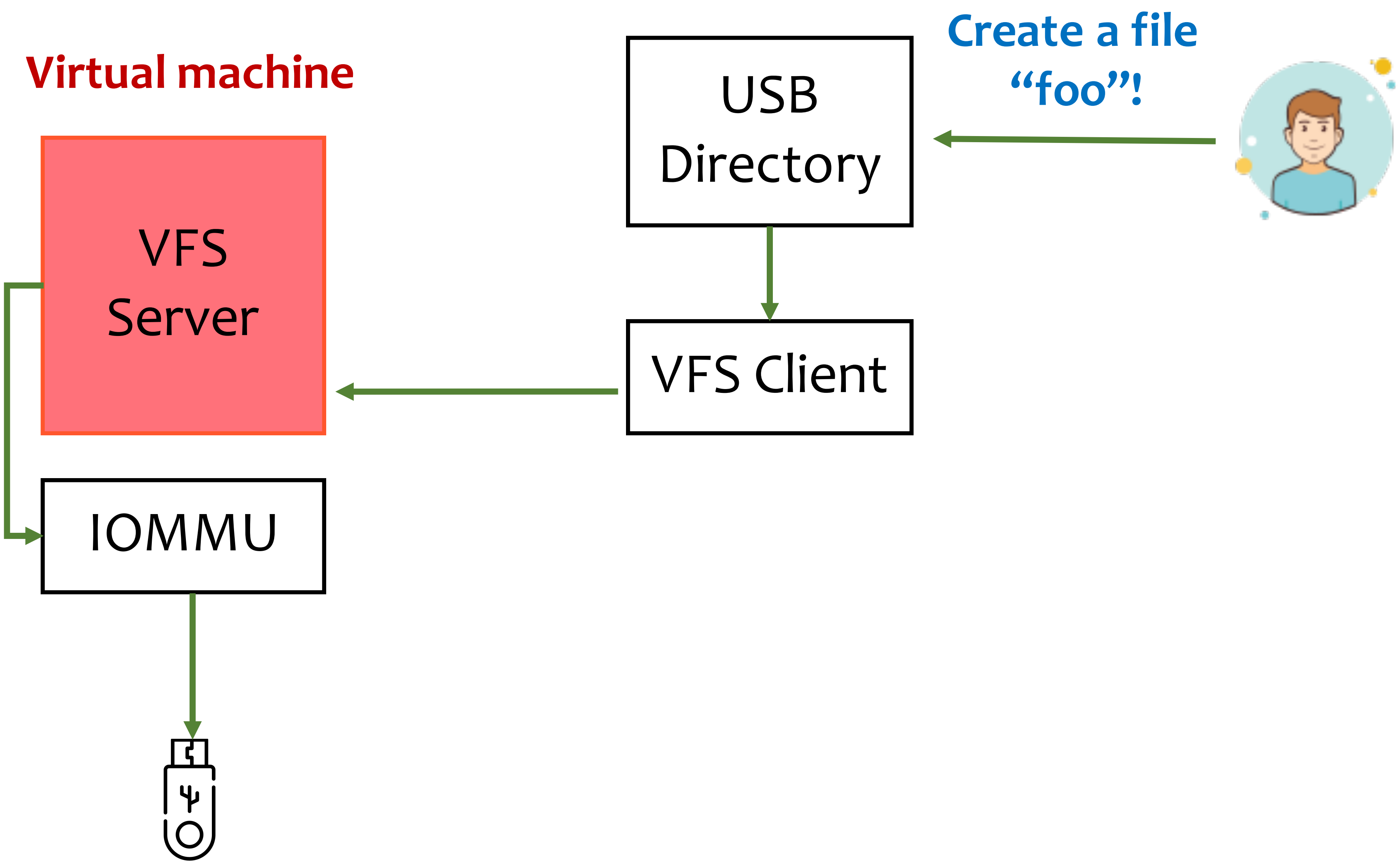
System overview



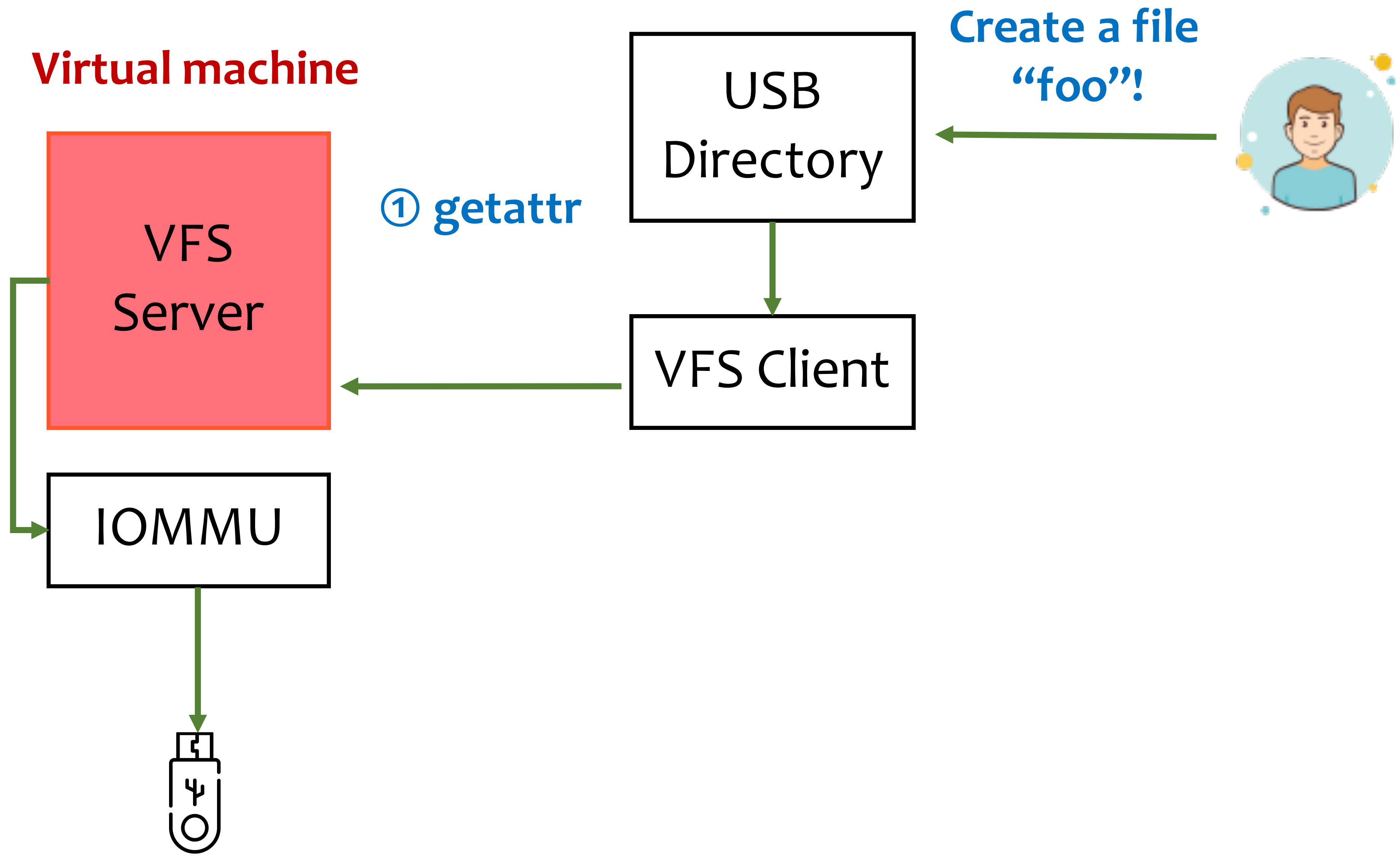
System overview



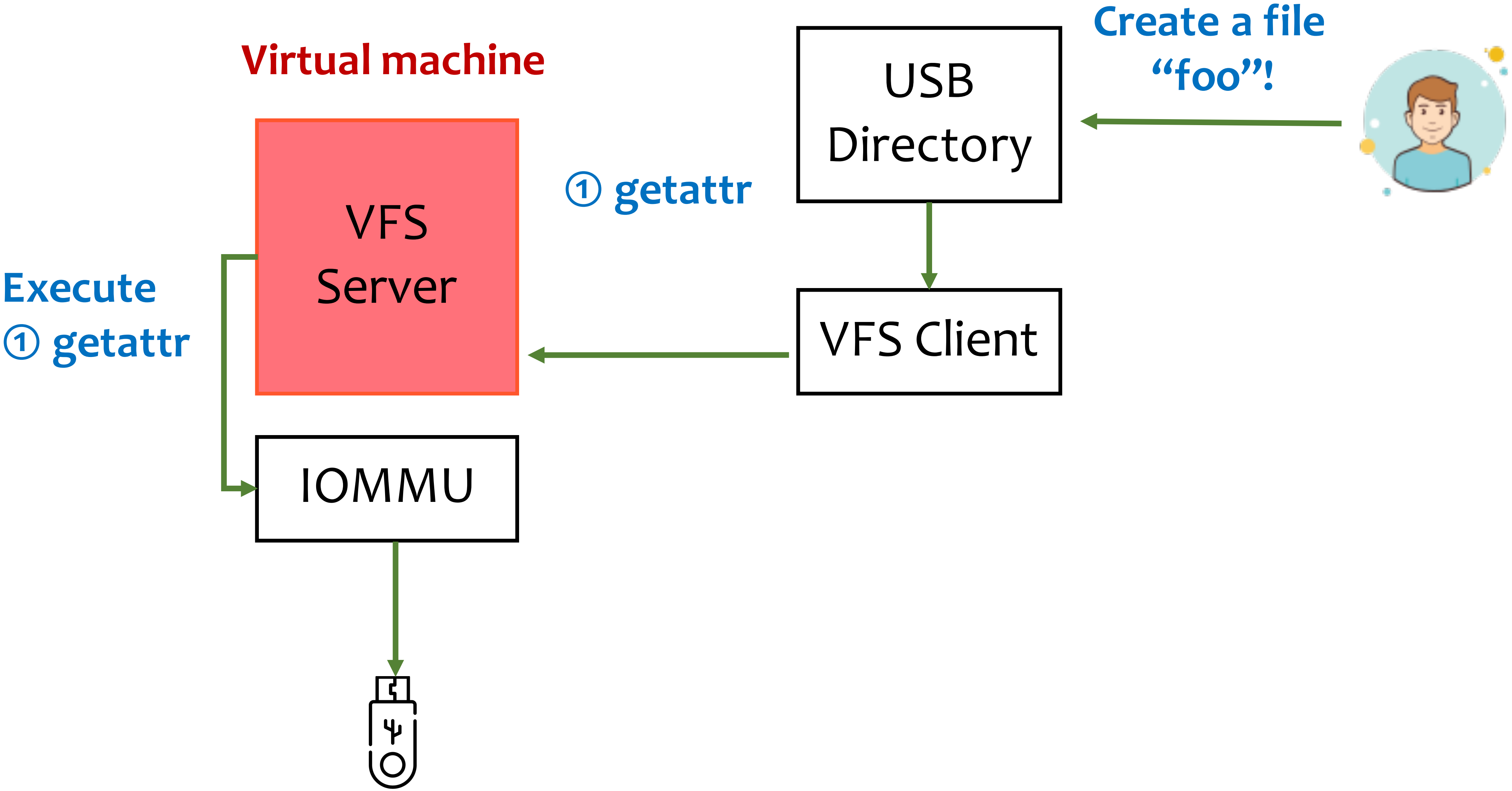
How *RBFuse* runs



How *RBFuse* runs



How *RBFuse* runs



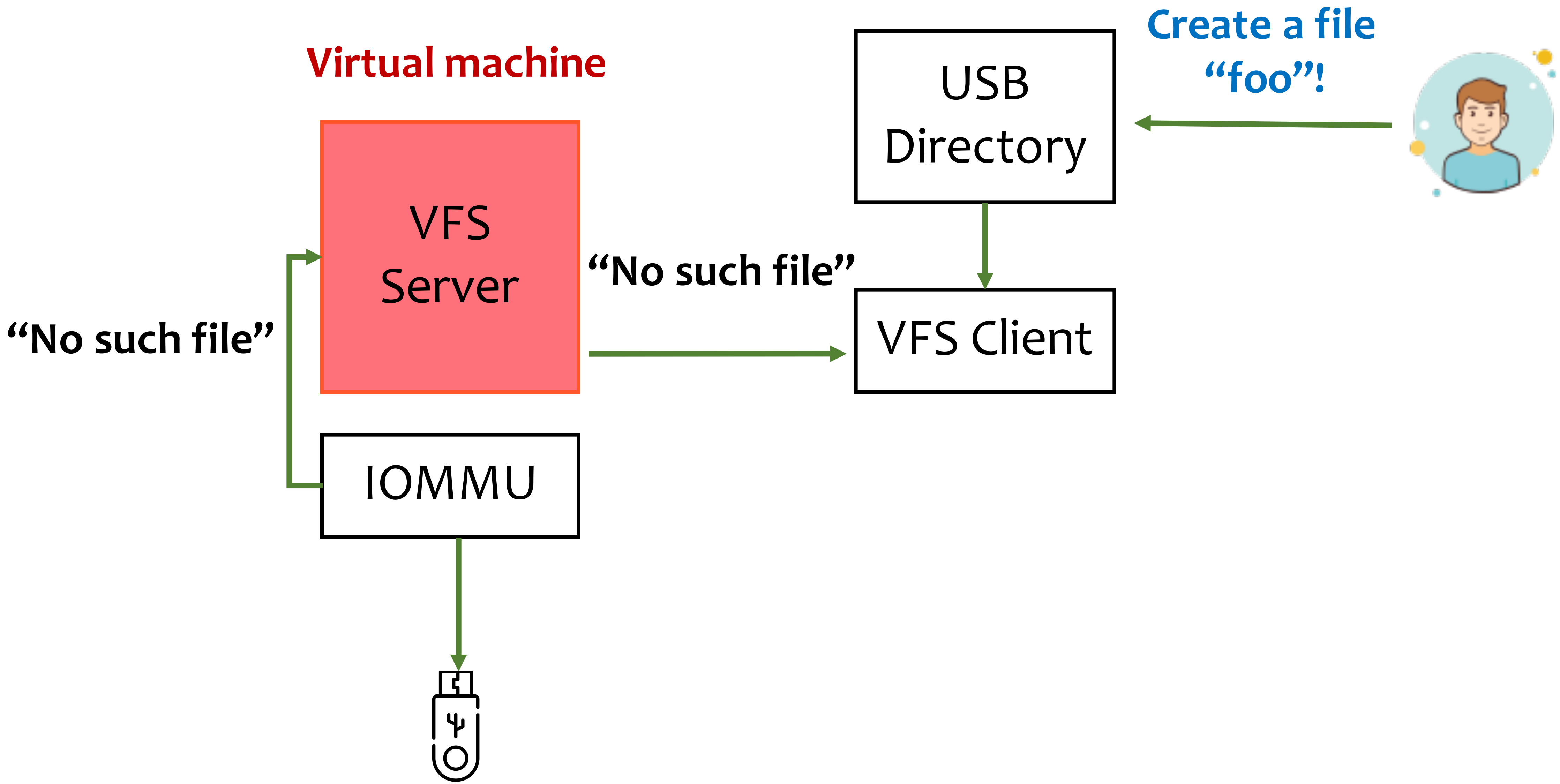
Virtual machine

Create a file "foo"!

① getattr

Execute ① getattr

How *RBFuse* runs



Create a file “foo”!

Virtual machine

USB Directory

VFS Server

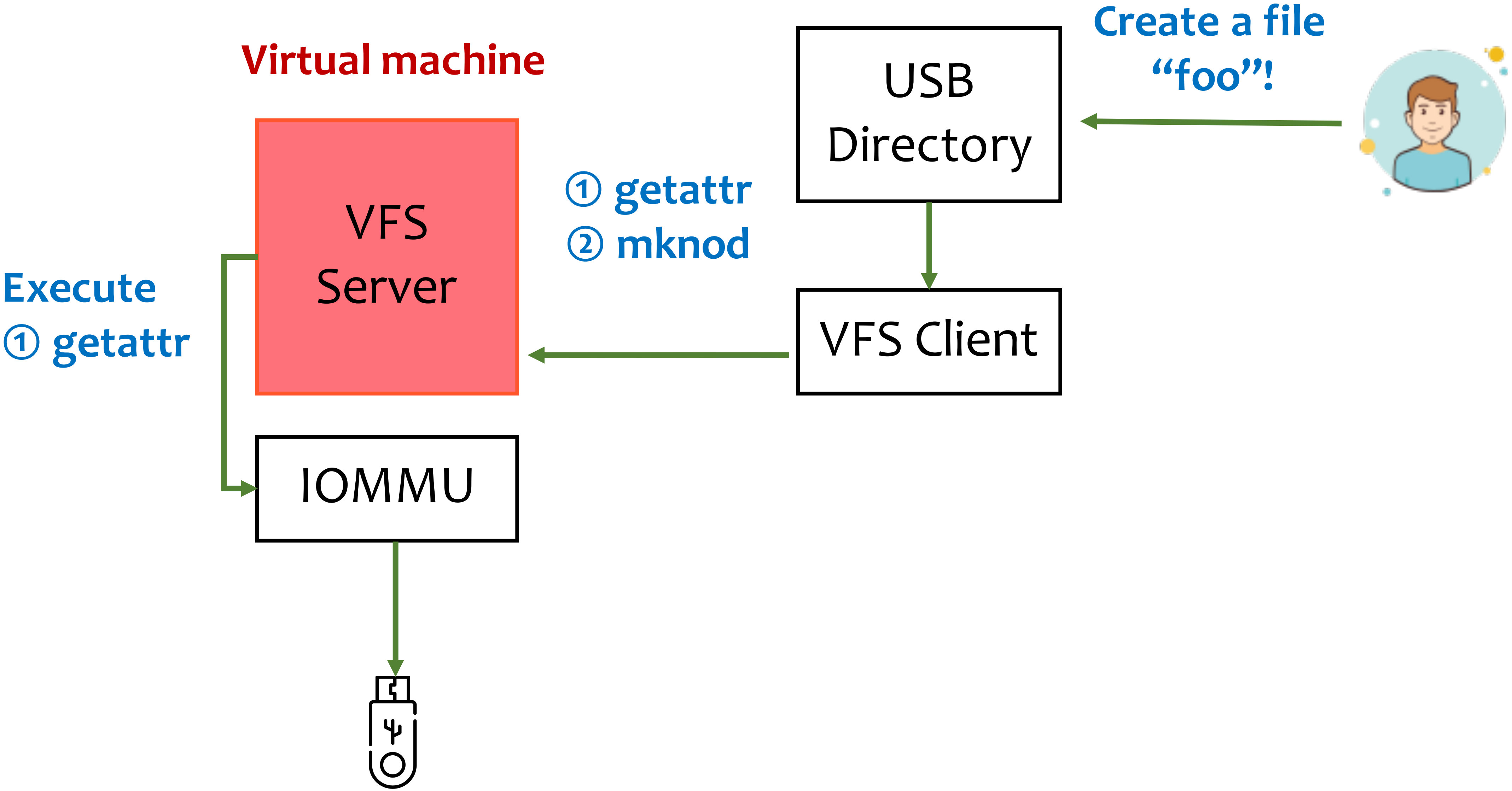
VFS Client

IOMMU

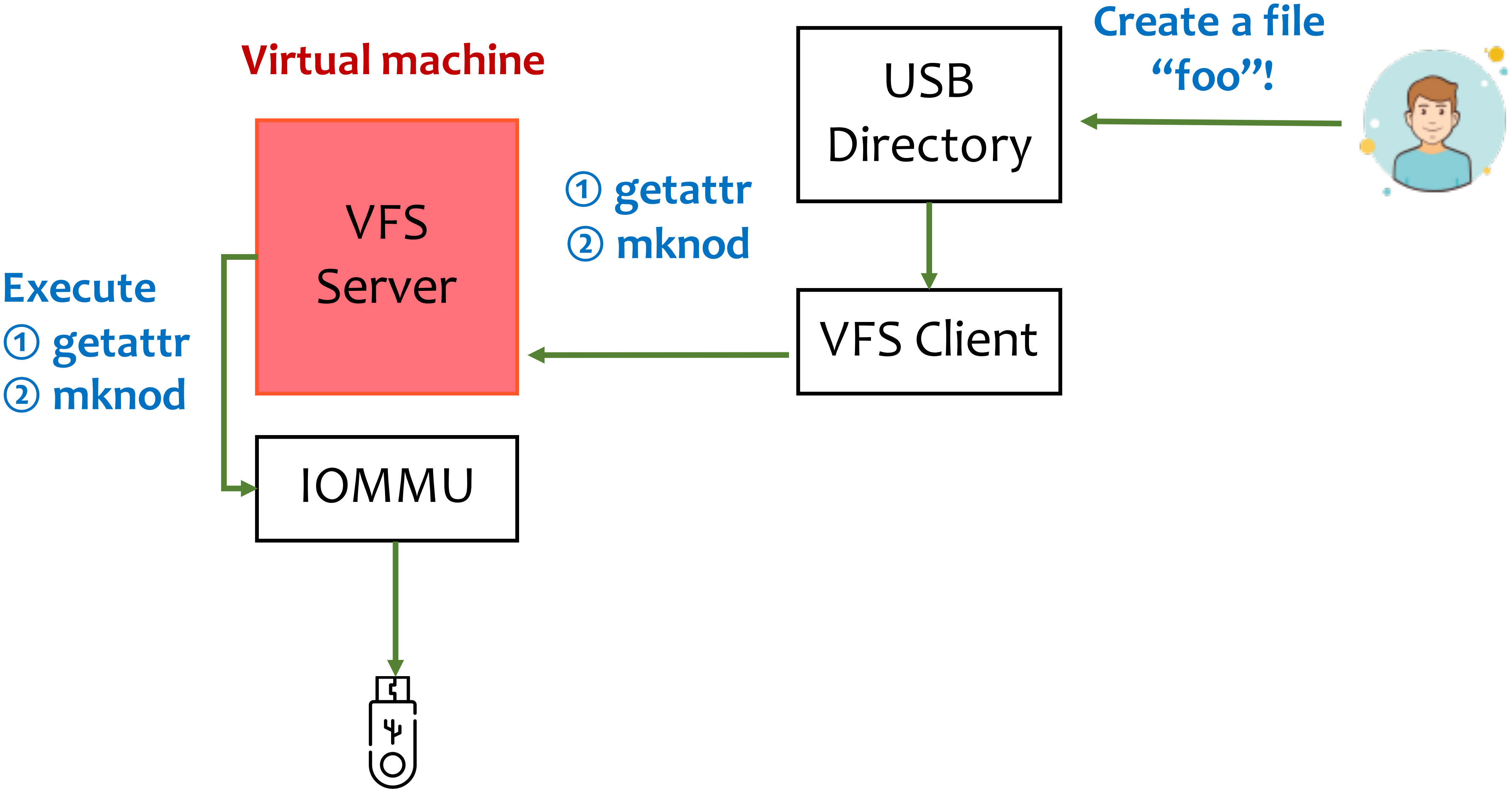
“No such file”

“No such file”

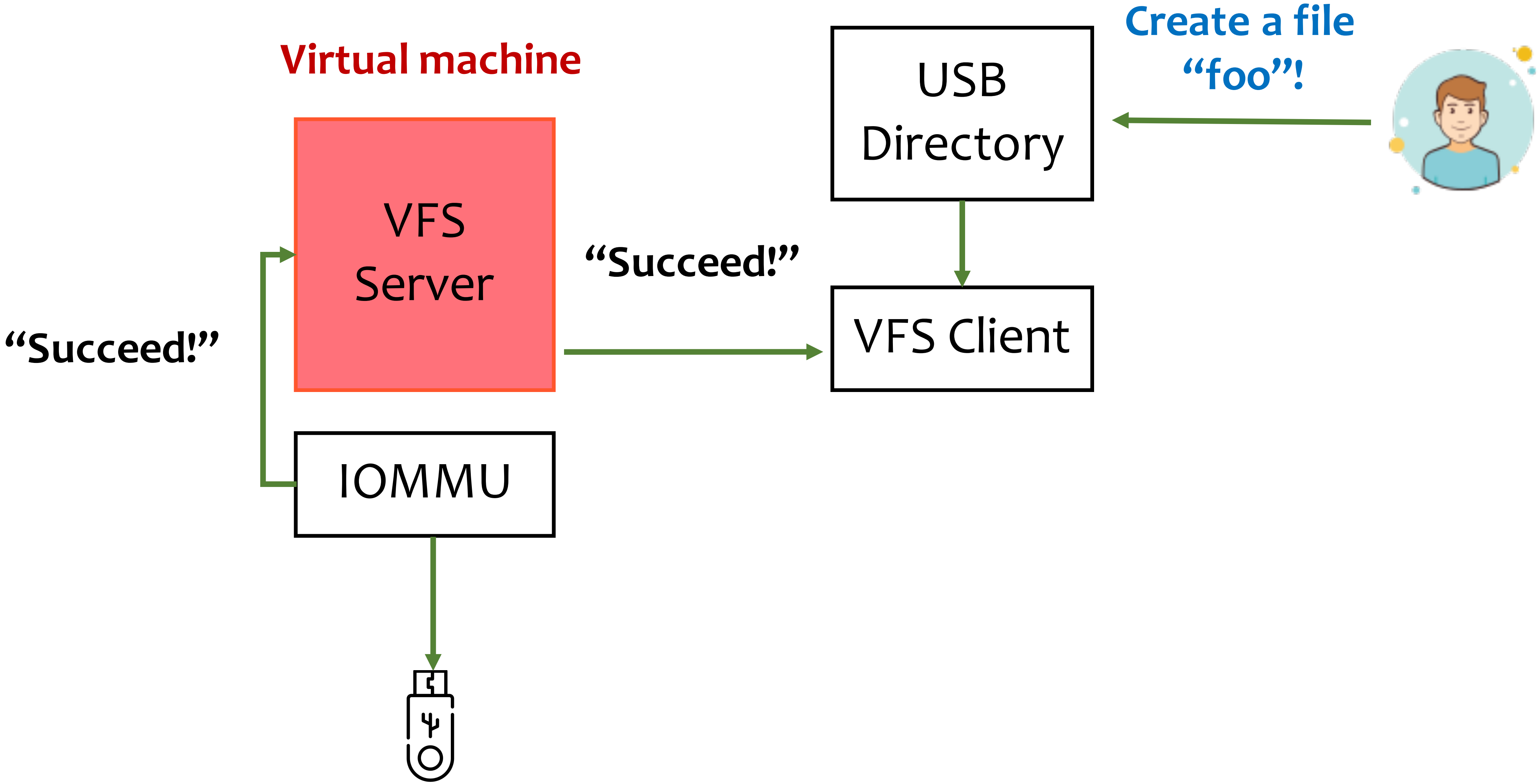
How *RBFuse* runs



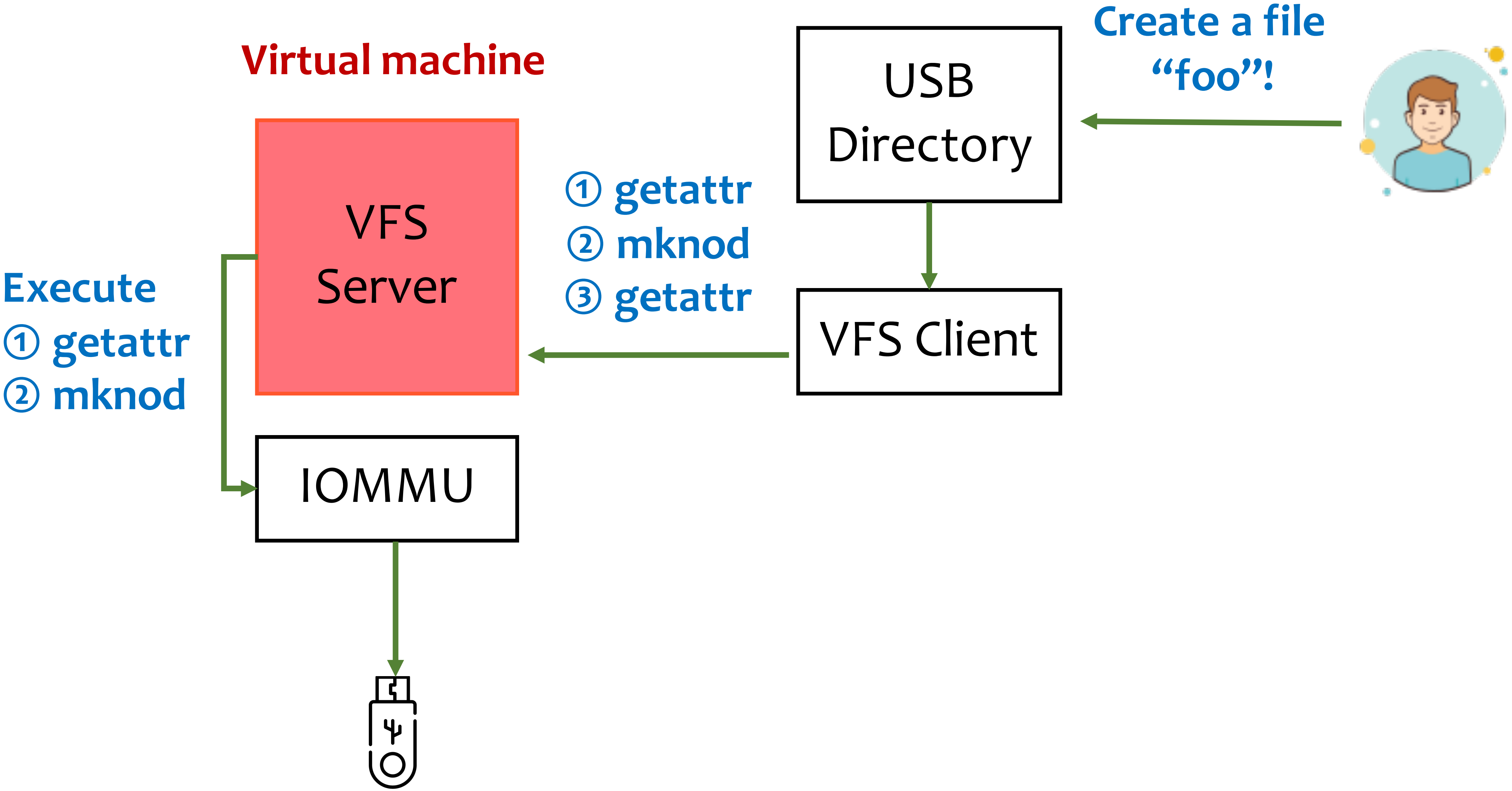
How *RBFuse* runs



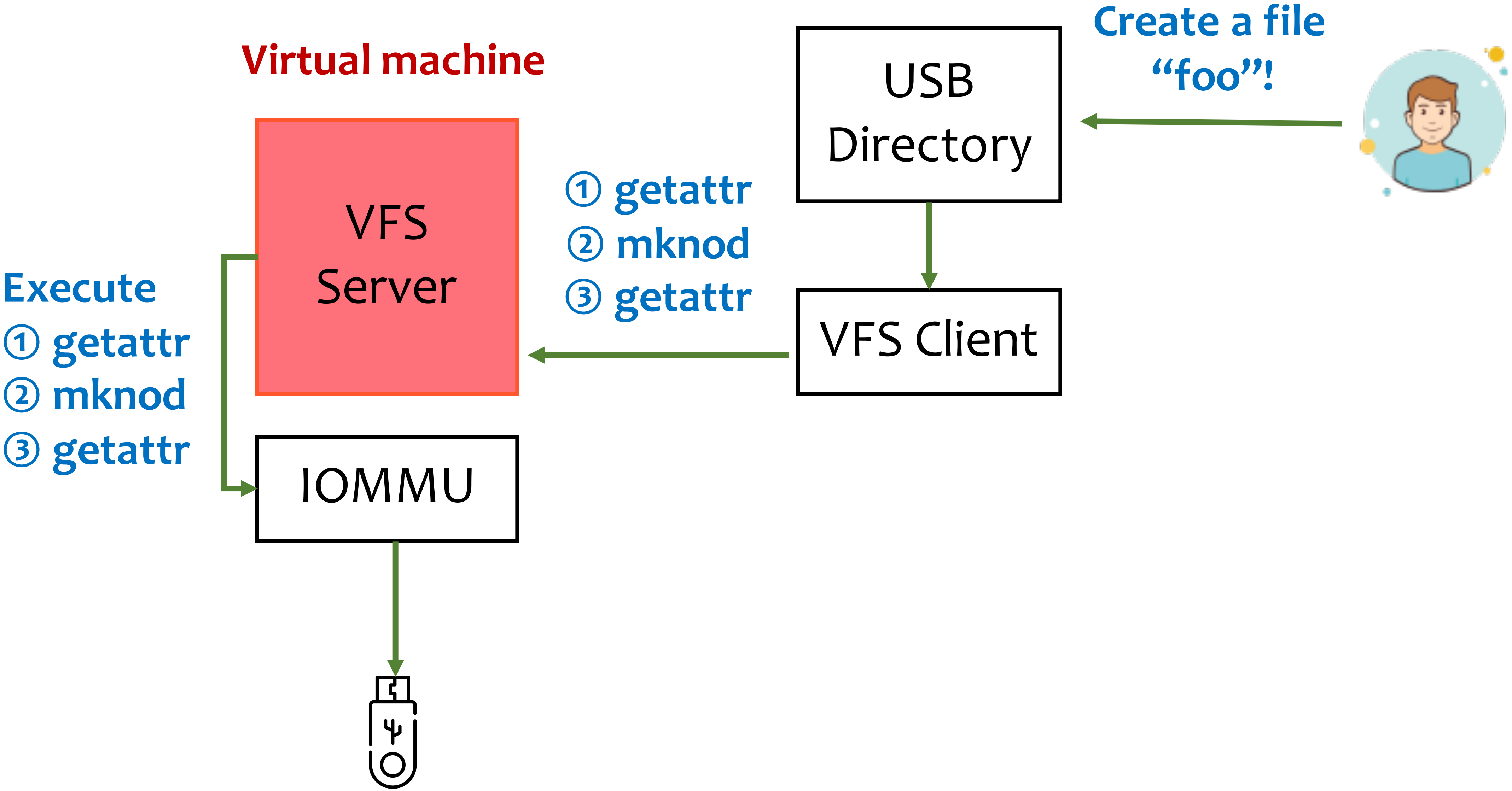
How *RBFuse* runs



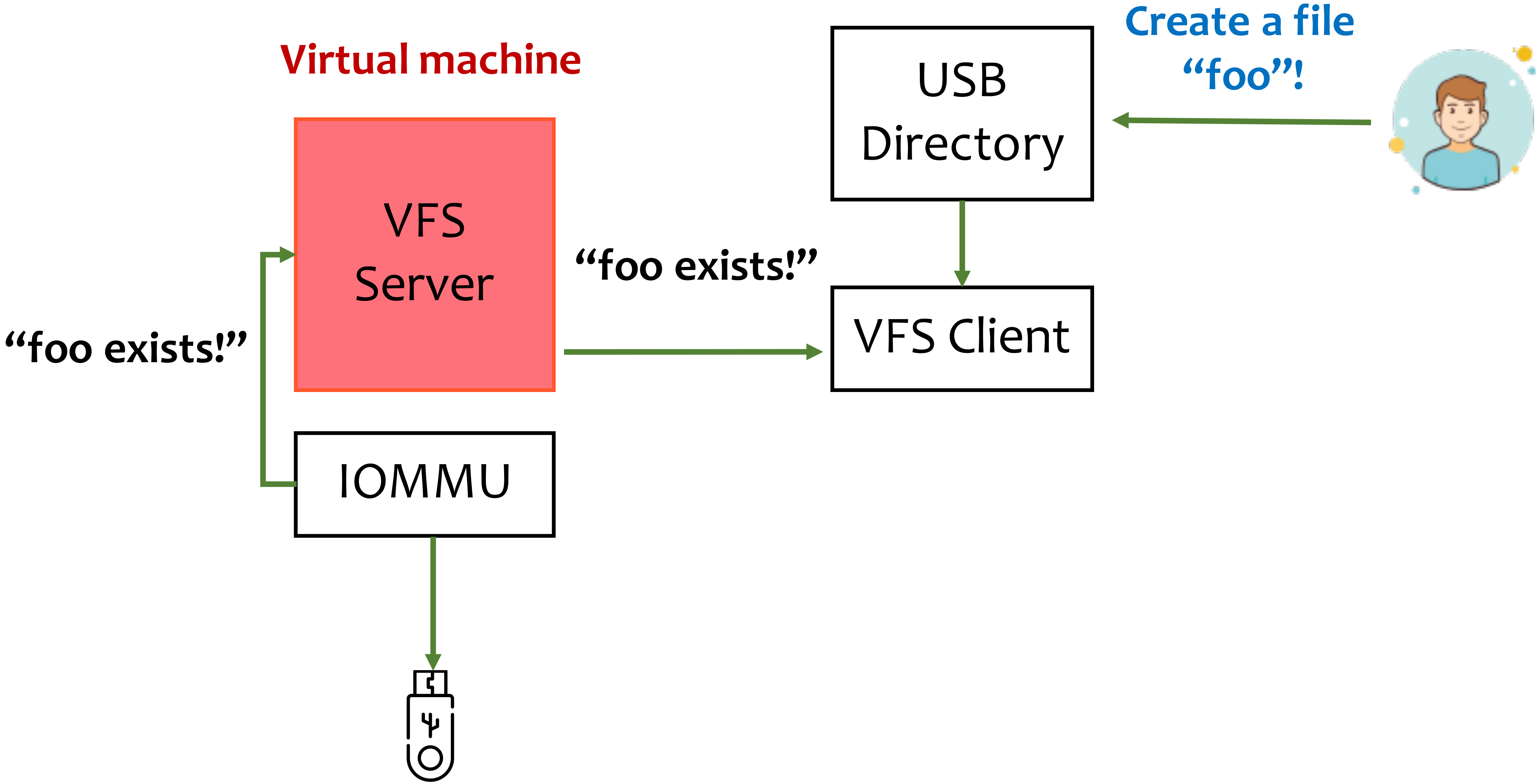
How *RBFuse* runs



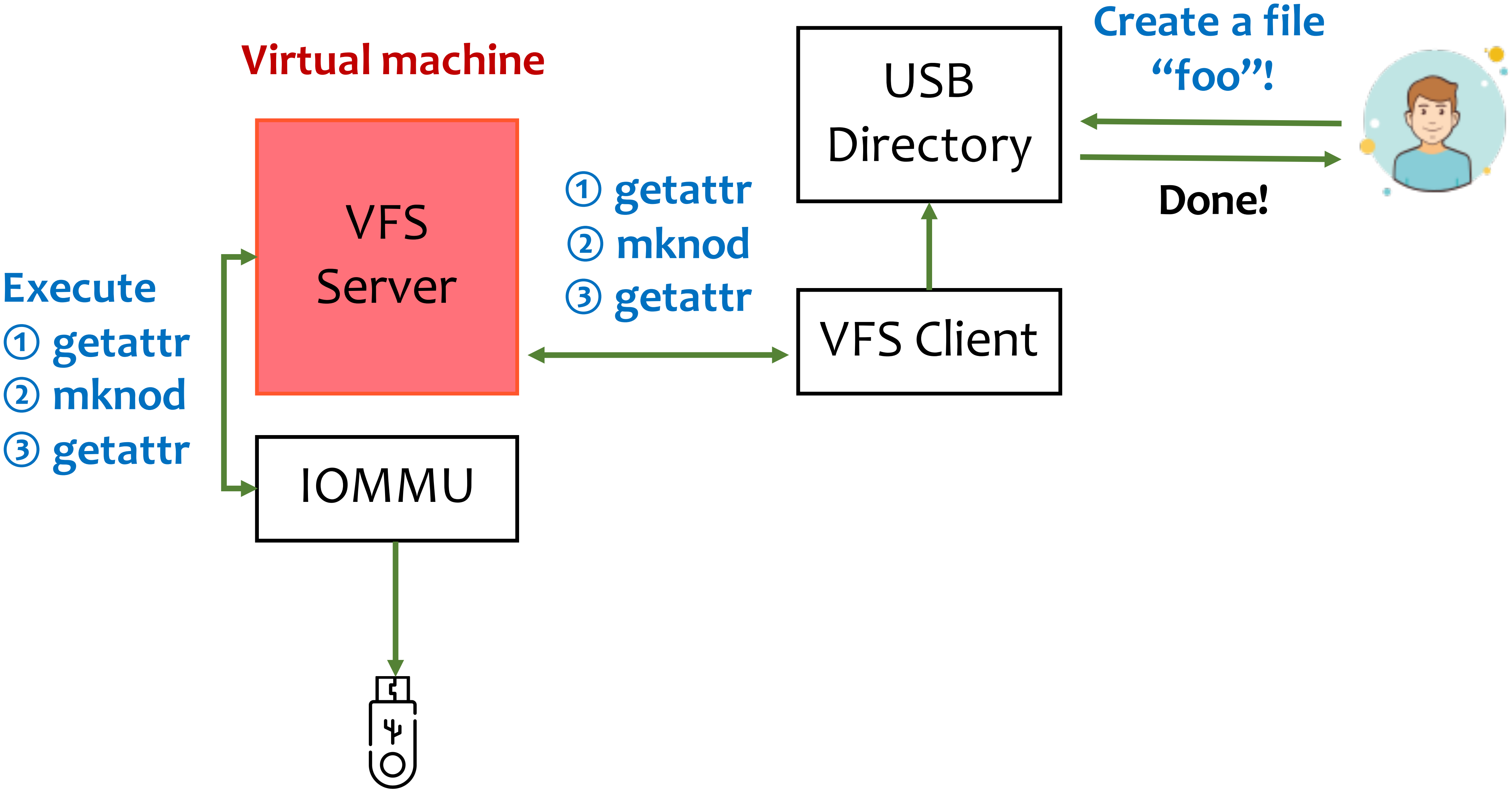
How *RBFuse* runs



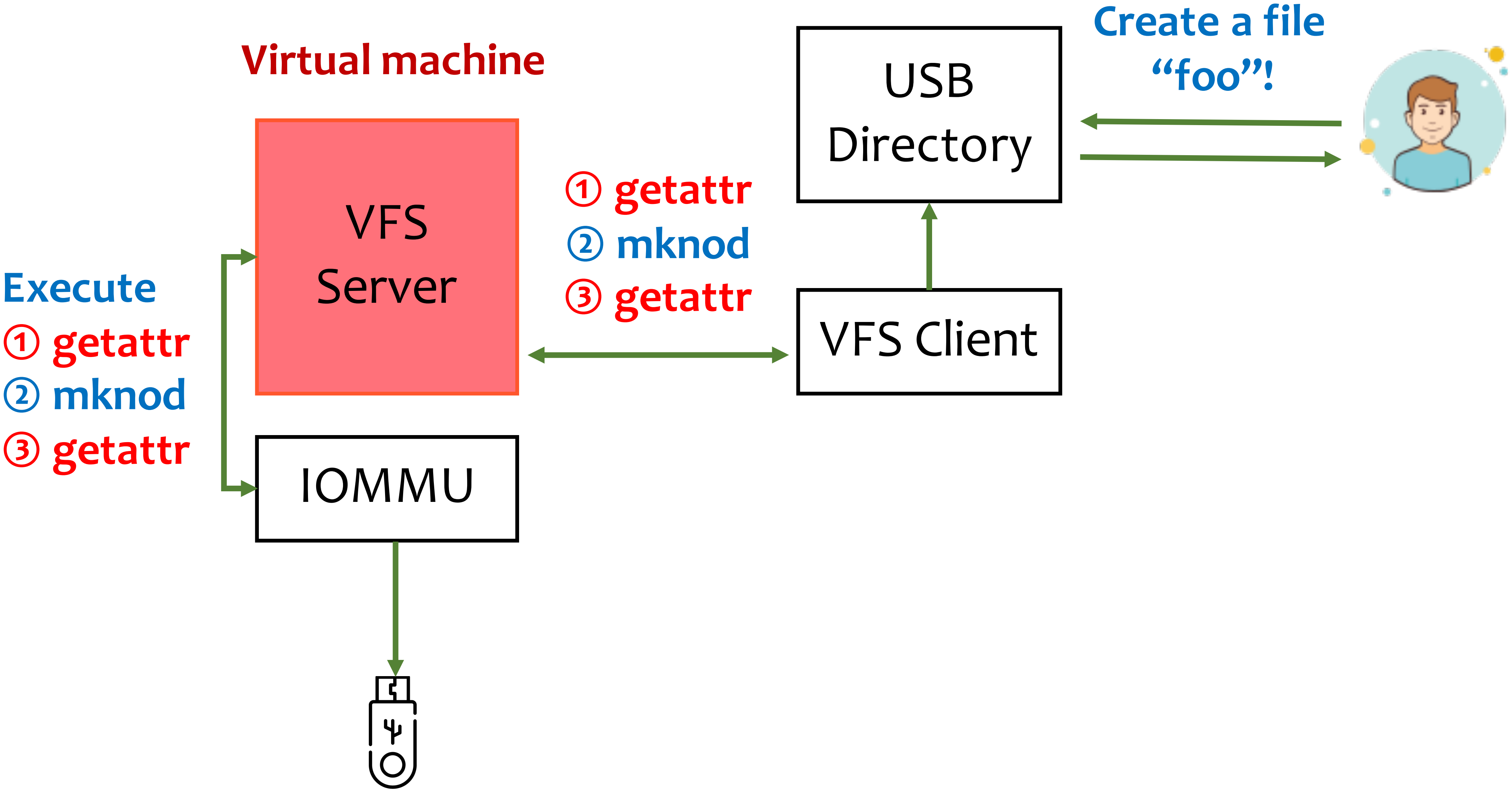
How *RBFuse* runs



How *RBFuse* runs



Performance issues



Virtual machine

VFS Server

IOMMU

Execute

① getattr

② mknod

③ getattr

① getattr

② mknod

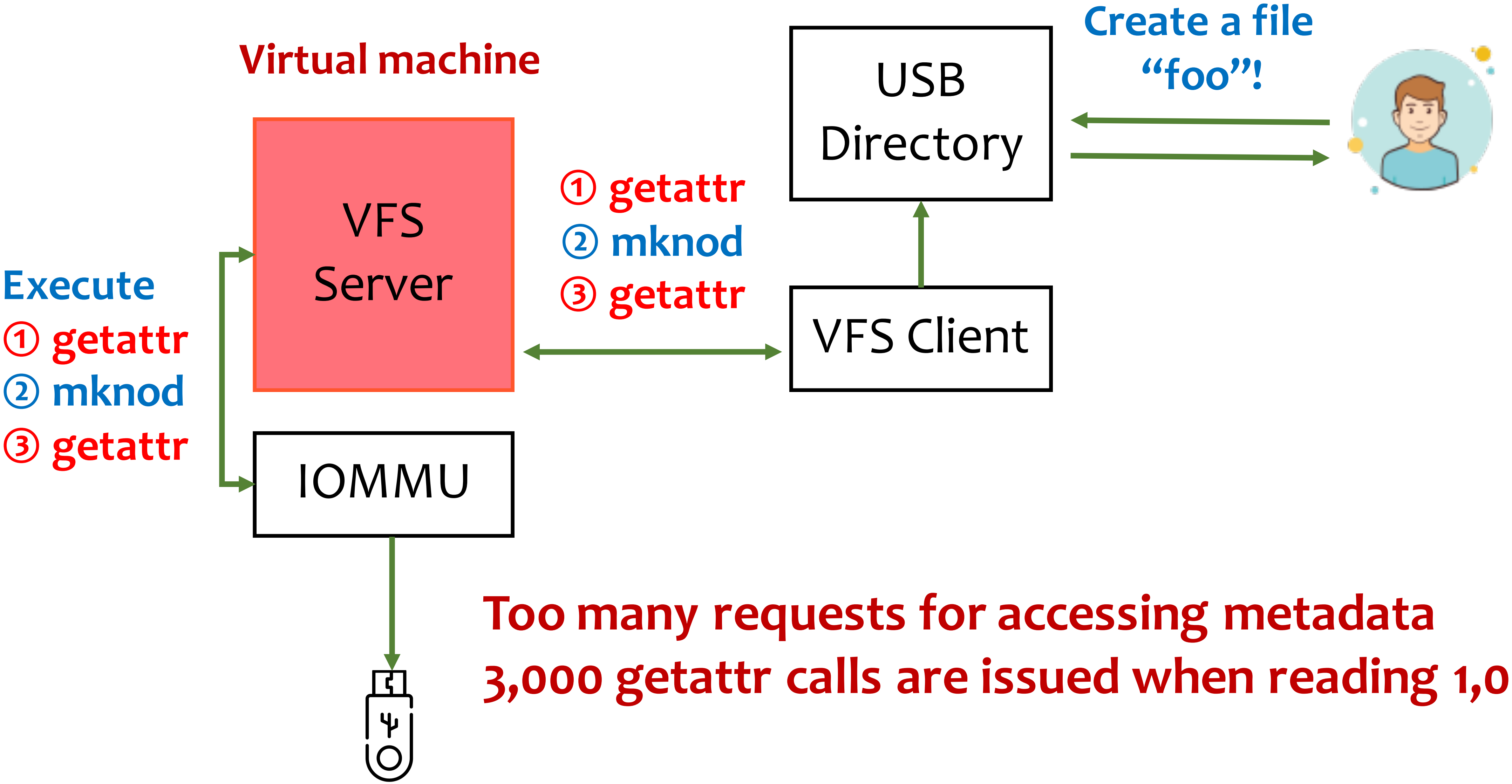
③ getattr

VFS Client

USB Directory

Create a file "foo"!

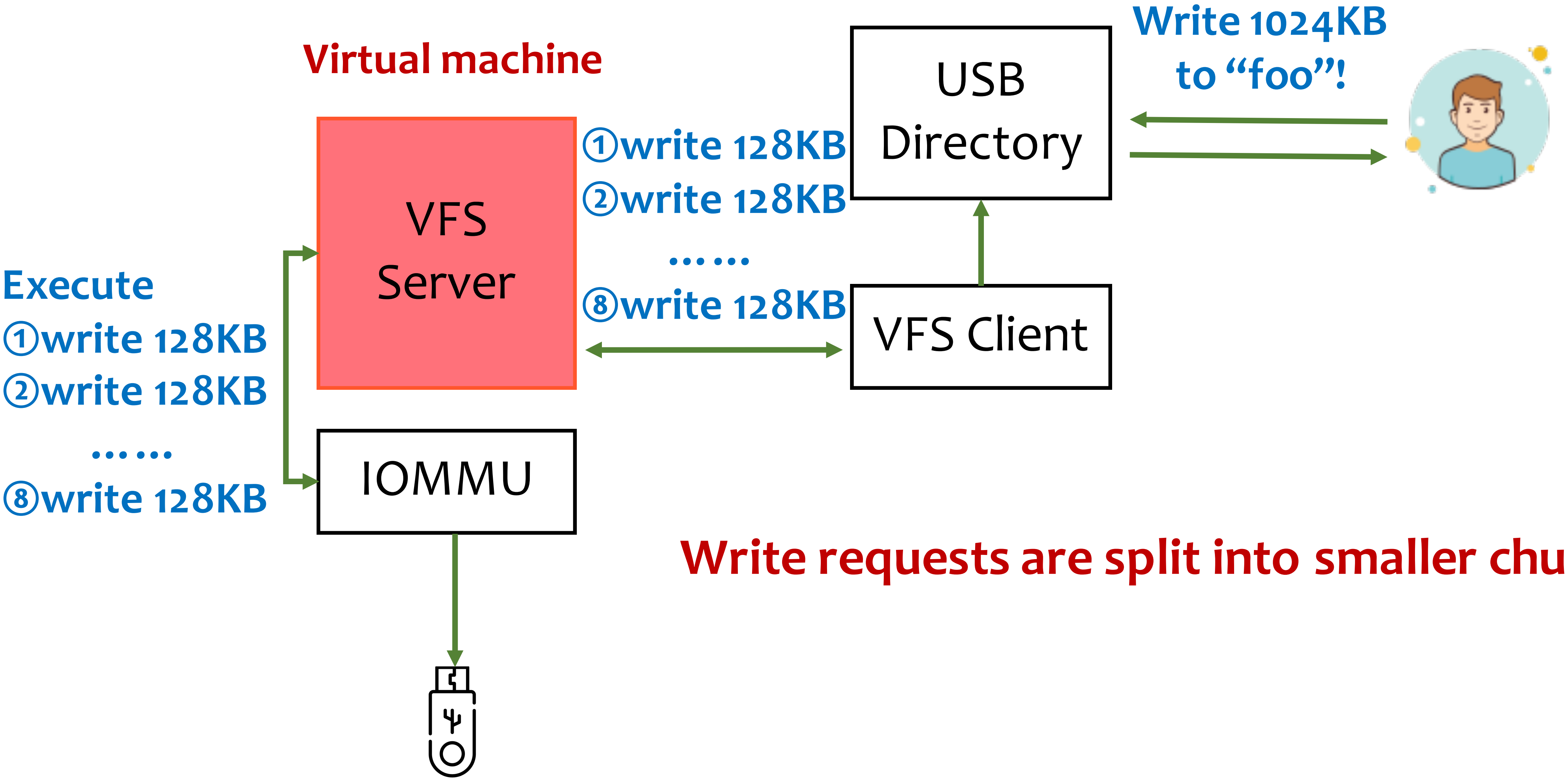
Performance issues



Too many requests for accessing metadata

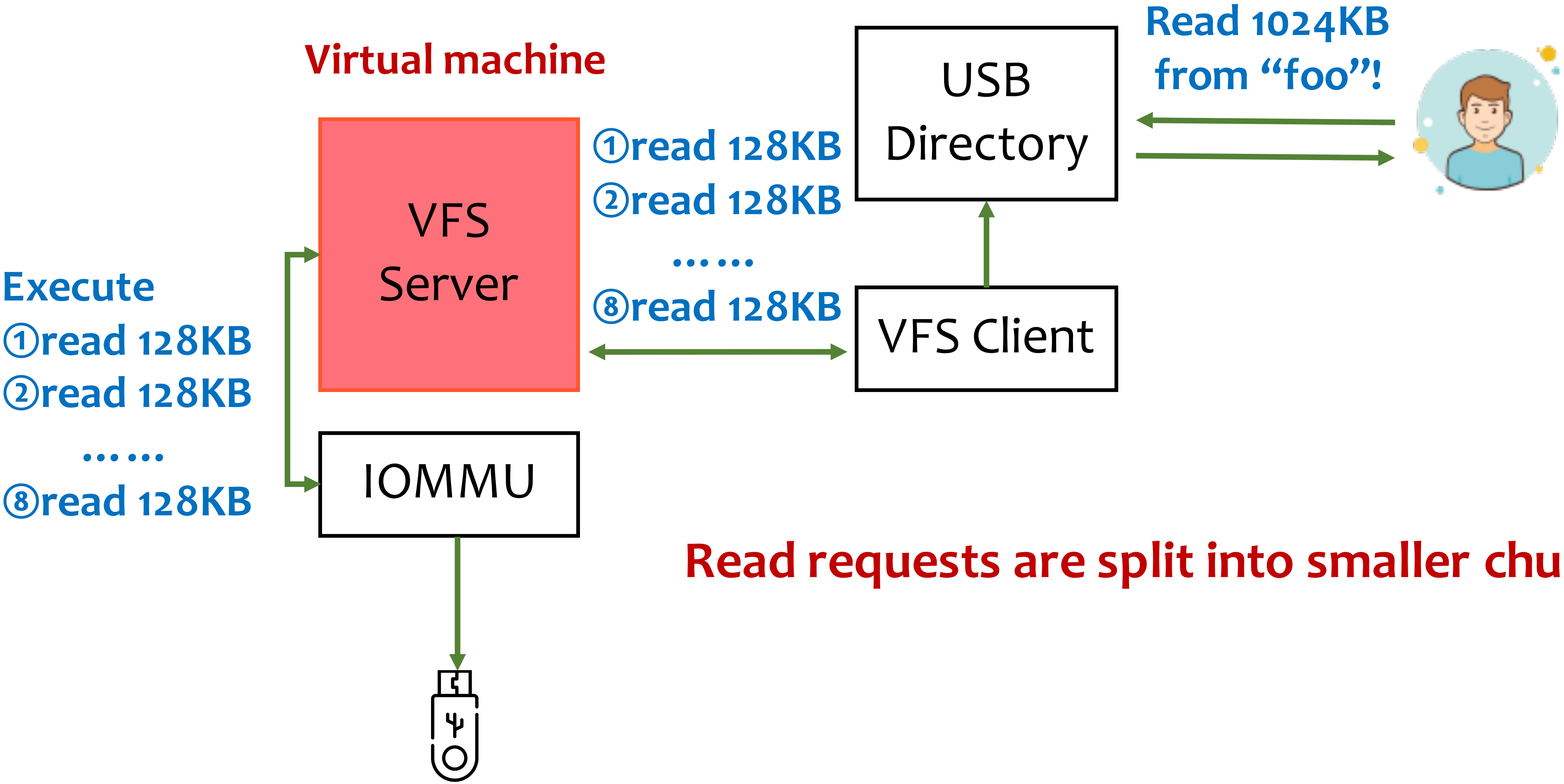
3,000 getattr calls are issued when reading 1,000 files

Performance issues



Write requests are split into smaller chunks

Performance issues



Virtual machine

VFS Server

IOMMU

Execute
①read 128KB
②read 128KB
……
⑧read 128KB

①read 128KB
②read 128KB
……
⑧read 128KB

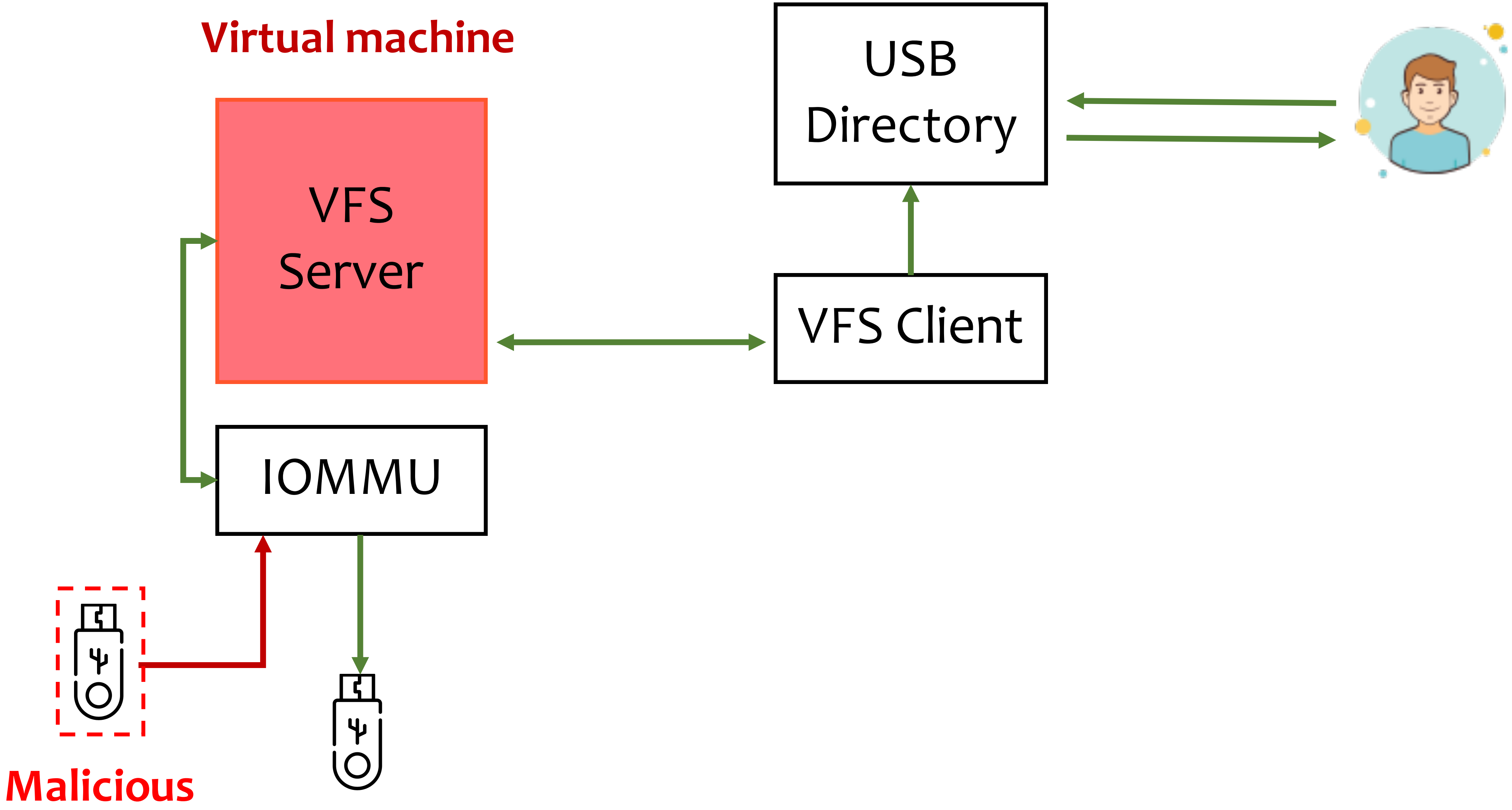
USB Directory

VFS Client

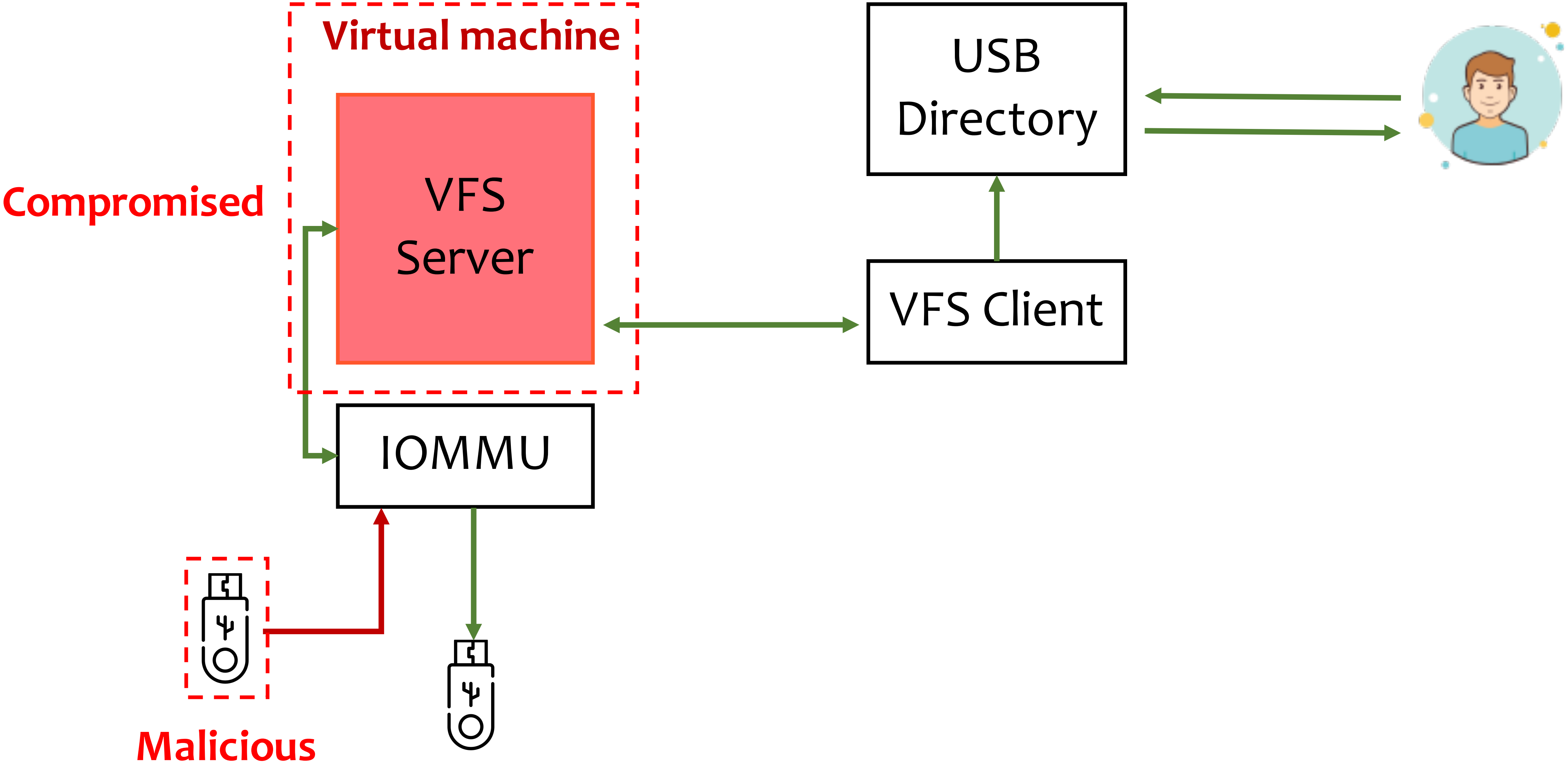
Read 1024KB from "foo"!

Read requests are split into smaller chunks

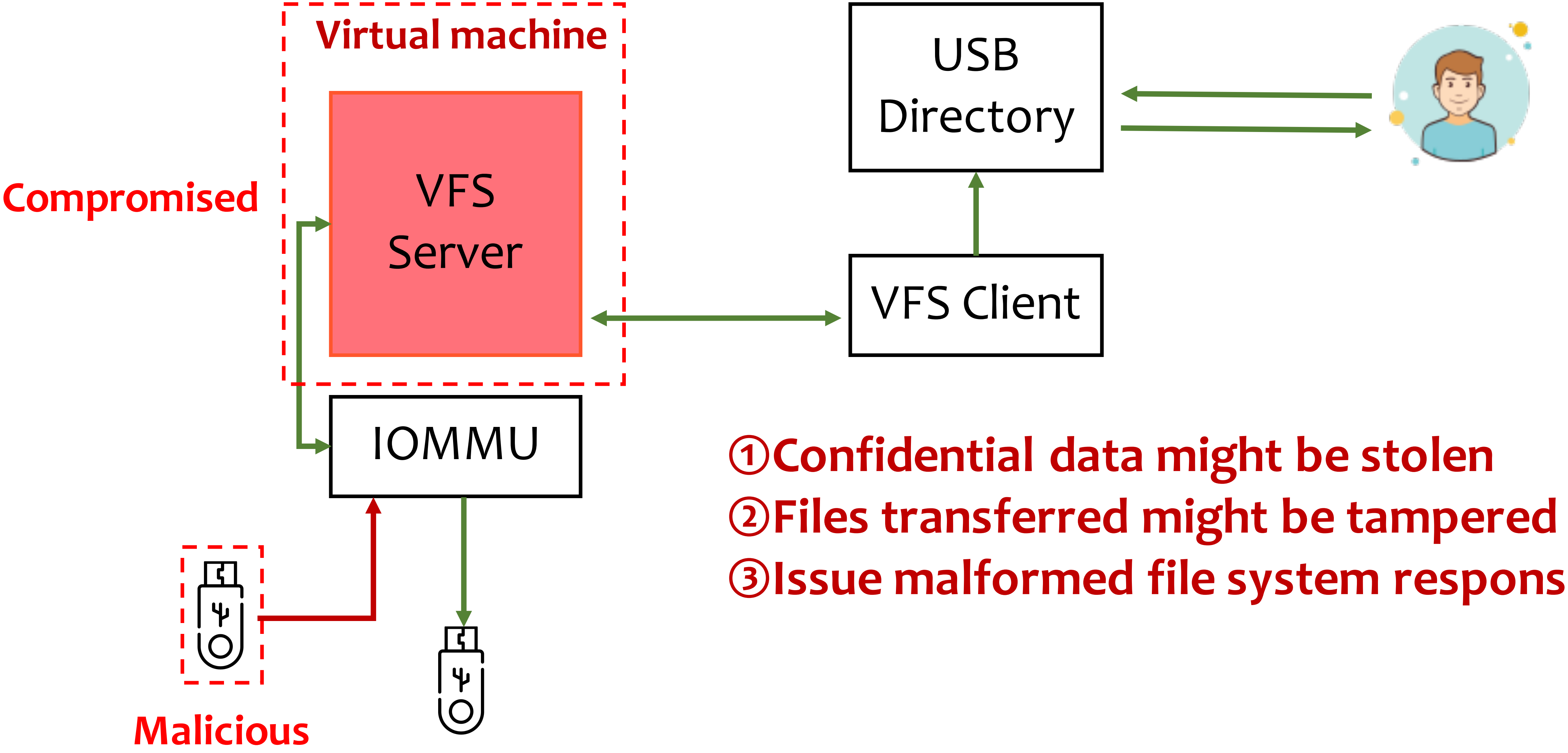
Compromised virtual machine



Compromised virtual machine



Compromised virtual machine

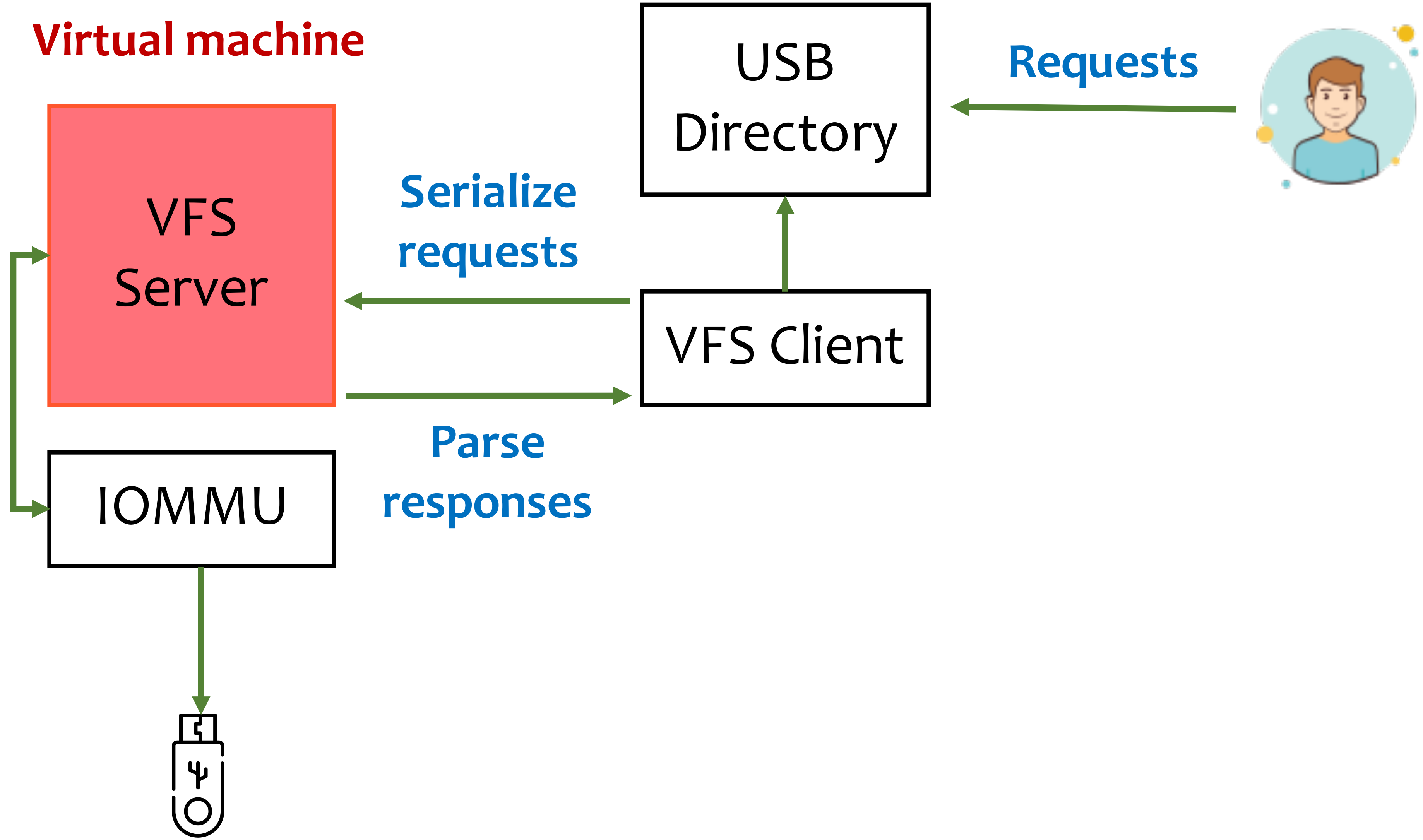


①Confidential data might be stolen

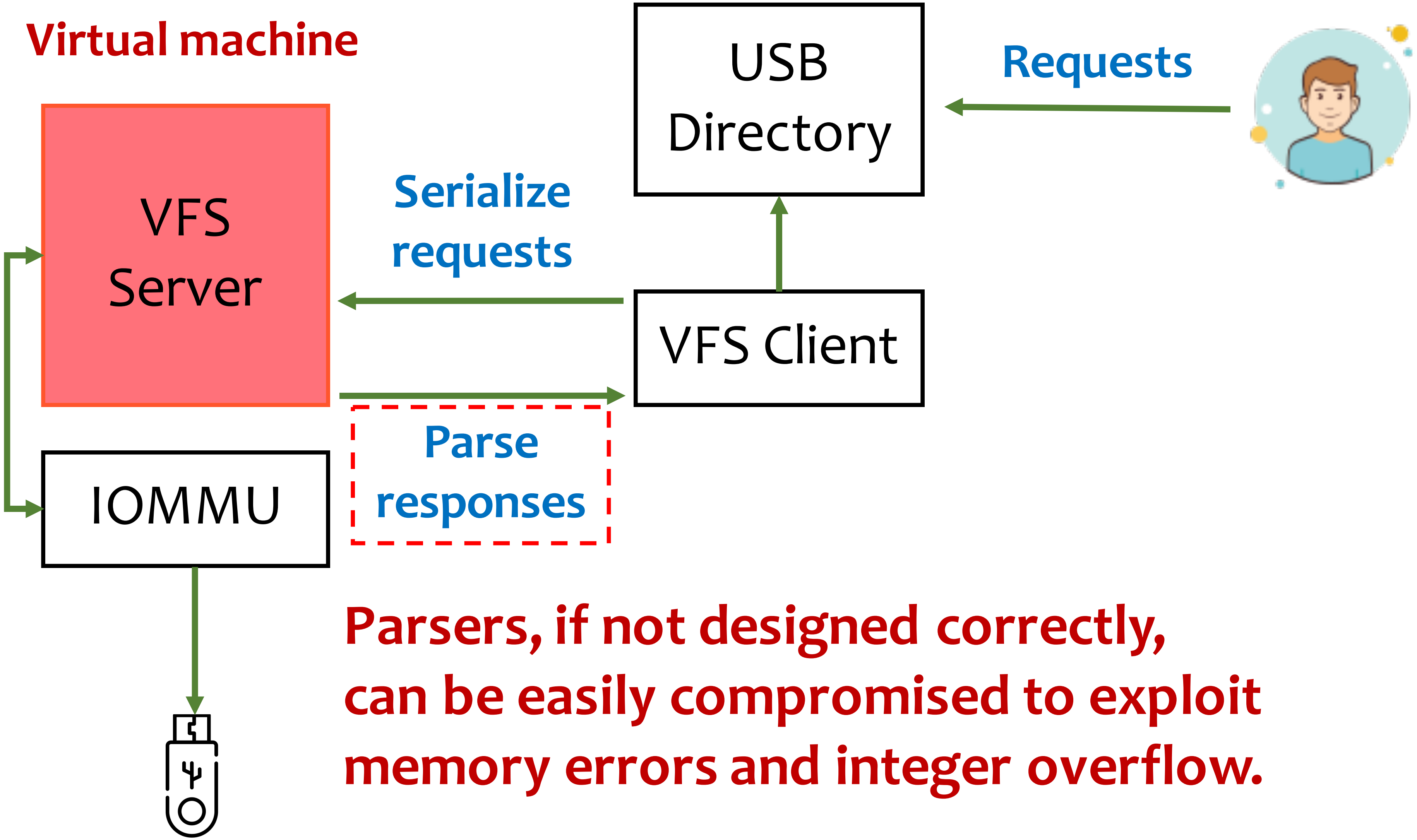
②Files transferred might be tampered

③Issue malformed file system responses

Parsing errors



Parsing errors



Parsers, if not designed correctly, can be easily compromised to exploit memory errors and integer overflow.

Agenda

- ◆ **How to address those challenges**
  - Optimizations
  - Encrypted communication
  - Formally verified serializer and parser

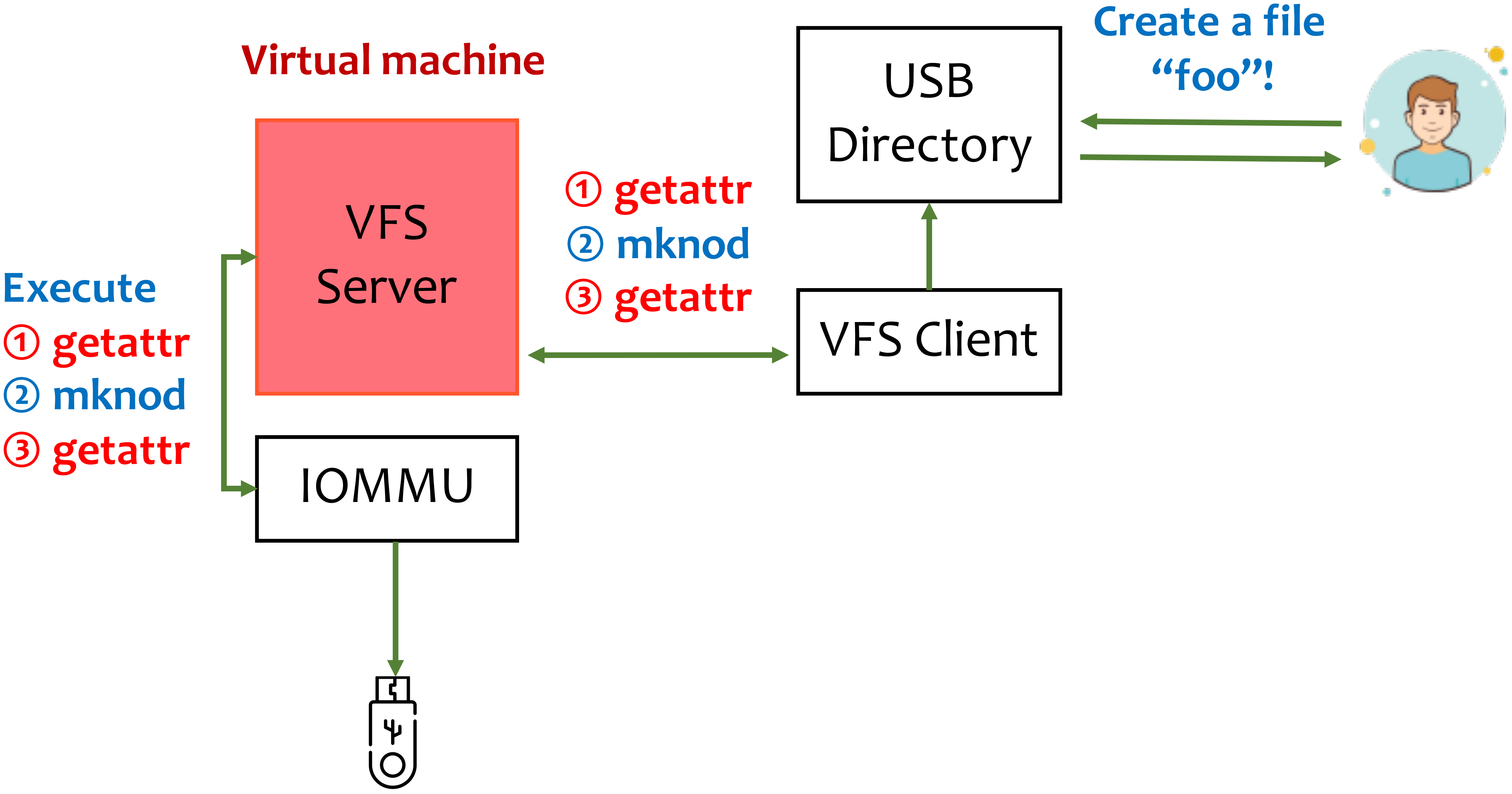
- ◆ **Preliminary evaluation**

- ◆ **Discussion & Conclusion**

Agenda

- ◆ **How to address those challenges**
  - Optimizations
  - Encrypted communication
  - Formally verified serializer and parser
- ◆ **Preliminary evaluation**
- ◆ **Discussion & Conclusion**

Caching metadata



Virtual machine

VFS Server

IOMMU

Execute
① getattr
② mknod
③ getattr

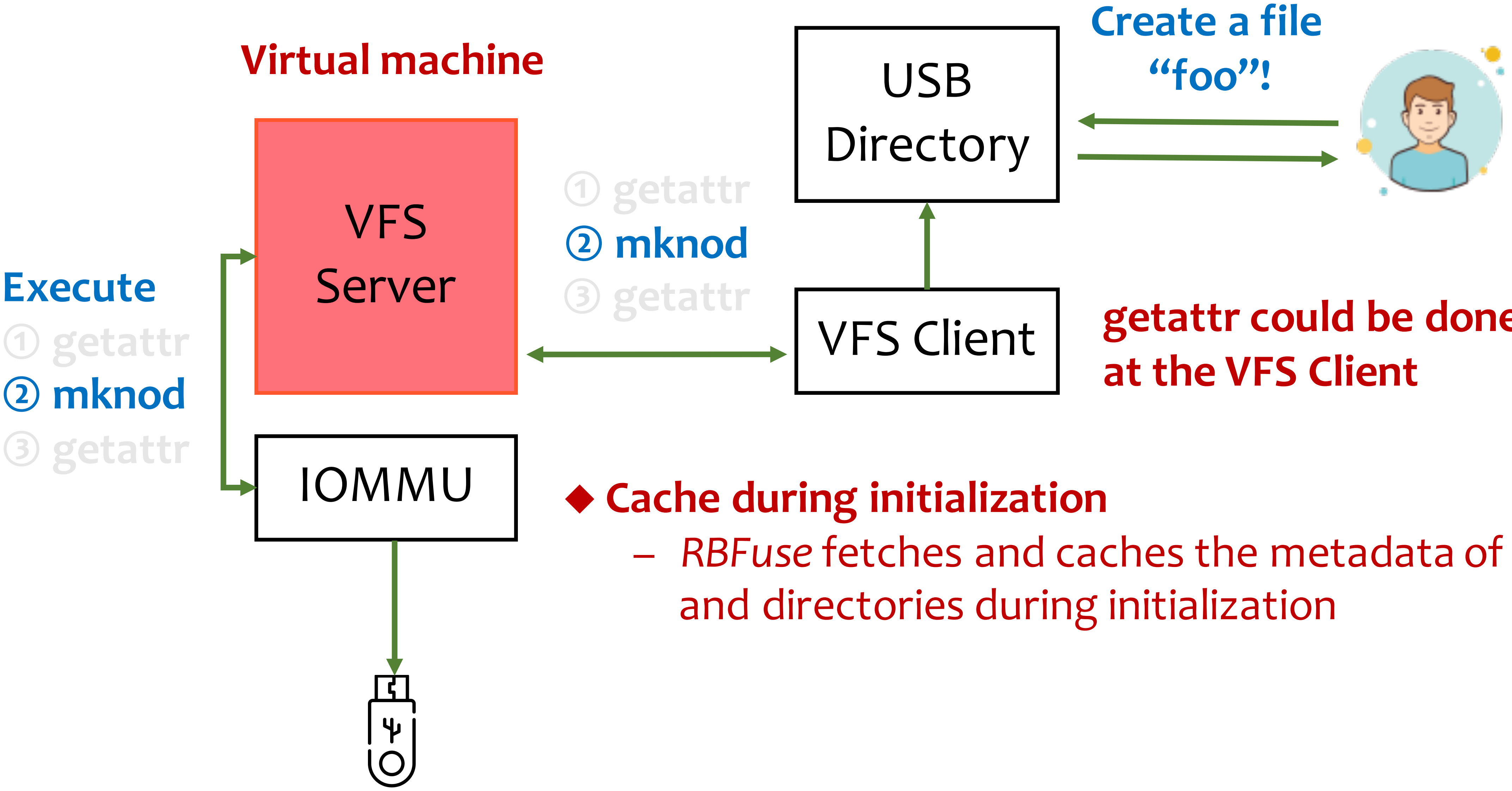
① getattr
② mknod
③ getattr

USB Directory

VFS Client

Create a file "foo"!

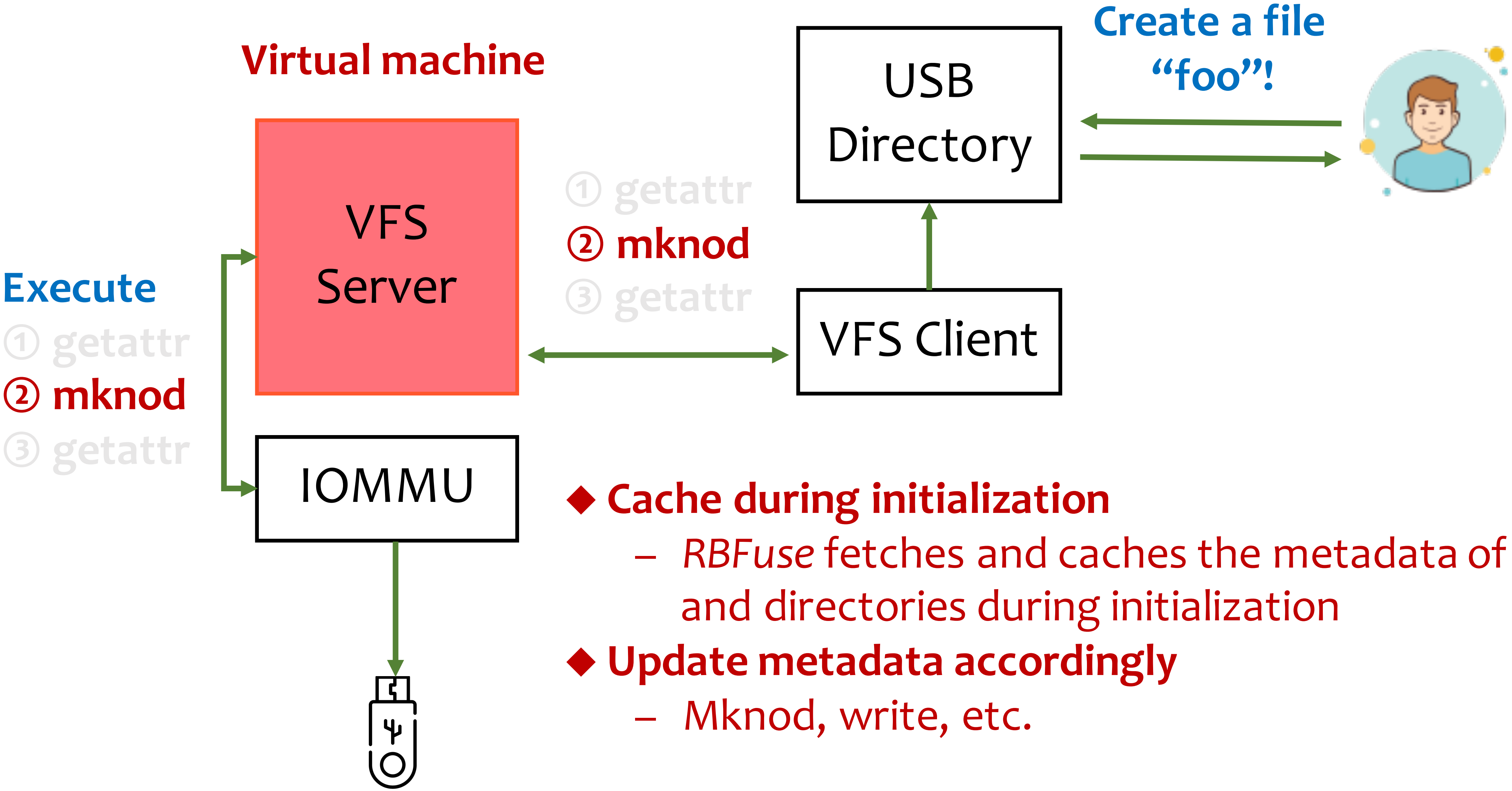
Caching metadata



getattr could be done locally at the VFS Client

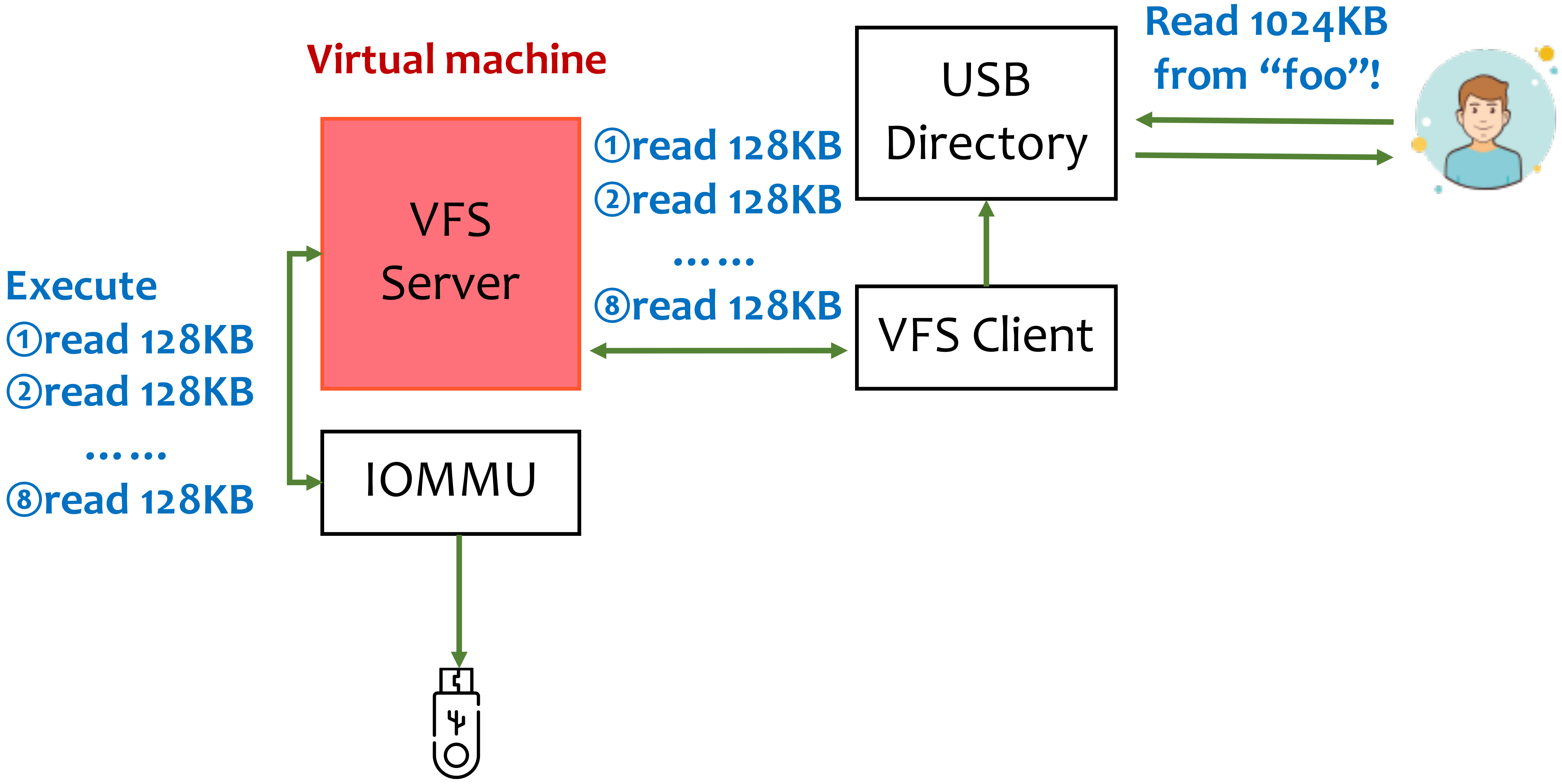
- **Cache during initialization**
  - *RBFuse* fetches and caches the metadata of all files and directories during initialization

Caching metadata



- ◆ **Cache during initialization**
  - – *RBFuse* fetches and caches the metadata of all files and directories during initialization
- ◆ **Update metadata accordingly**
  - – Mknod, write, etc.

Prefetching



Read 1024KB from “foo”!

USB Directory

VFS Client

①read 128KB

②read 128KB

……

⑧read 128KB

Virtual machine

VFS Server

IOMMU

Execute

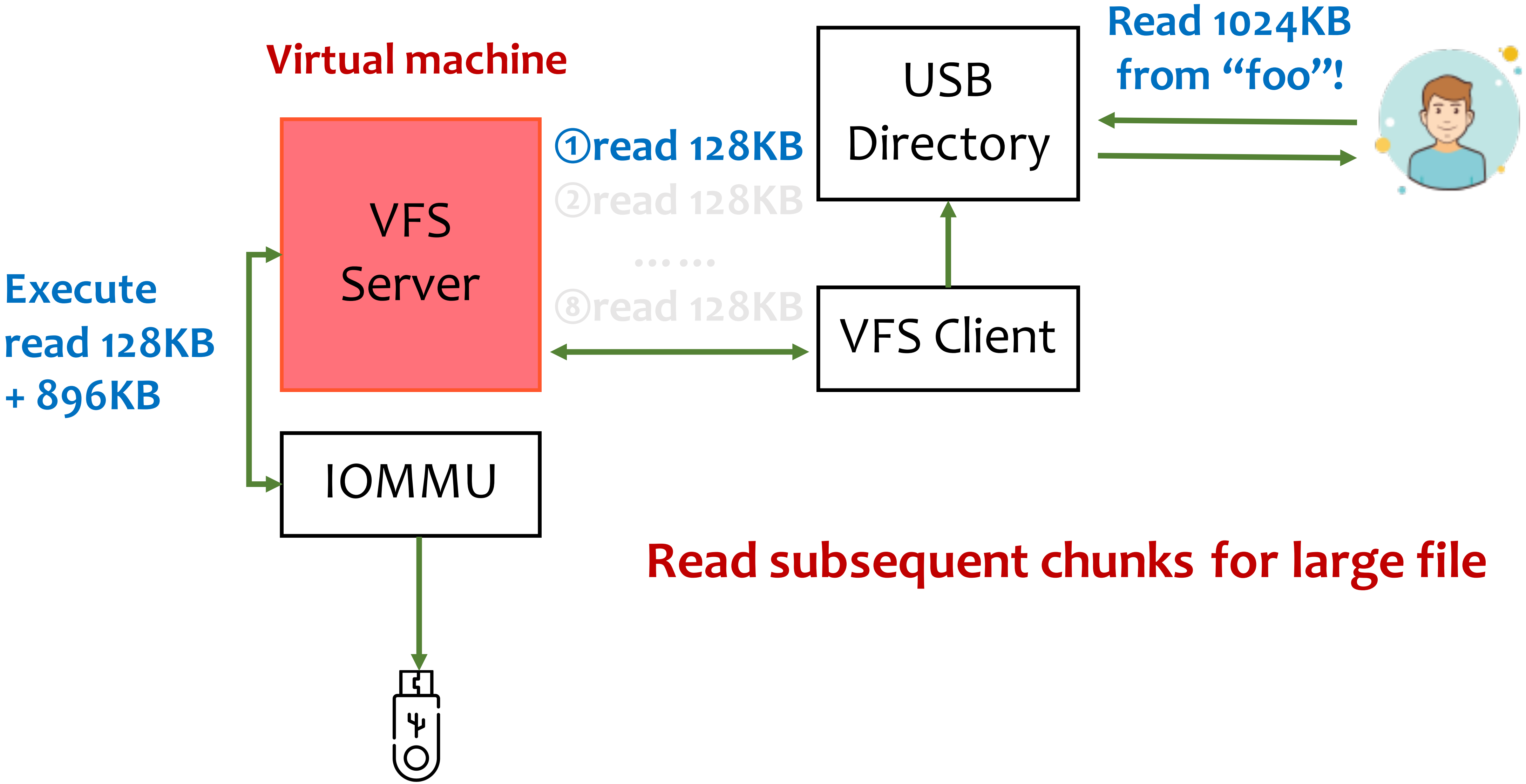
①read 128KB

②read 128KB

……

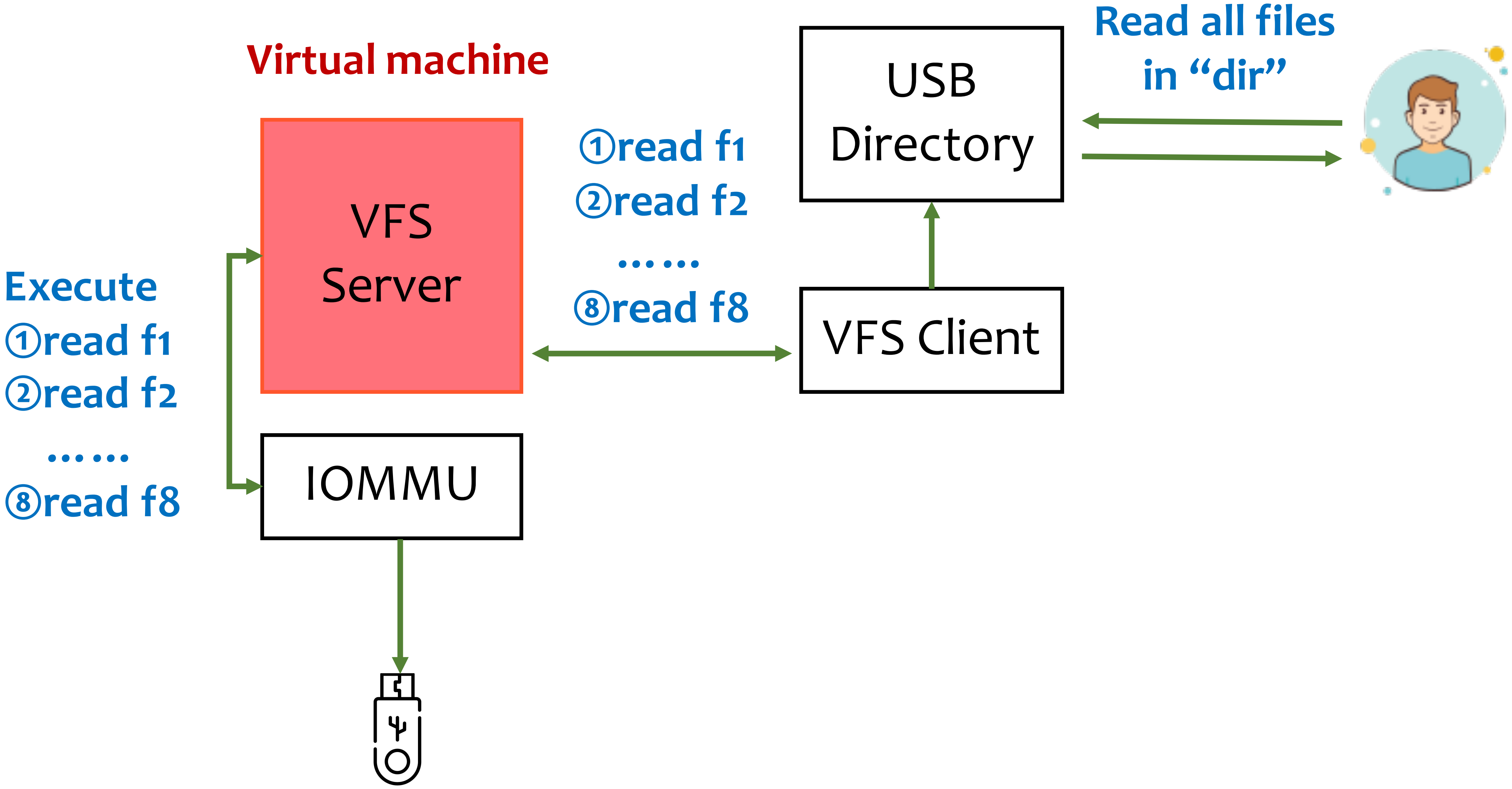
⑧read 128KB

Prefetching



Read subsequent chunks for large file

Prefetching



Virtual machine

VFS Server

IOMMU

Execute
①read f1
②read f2
……
⑧read f8

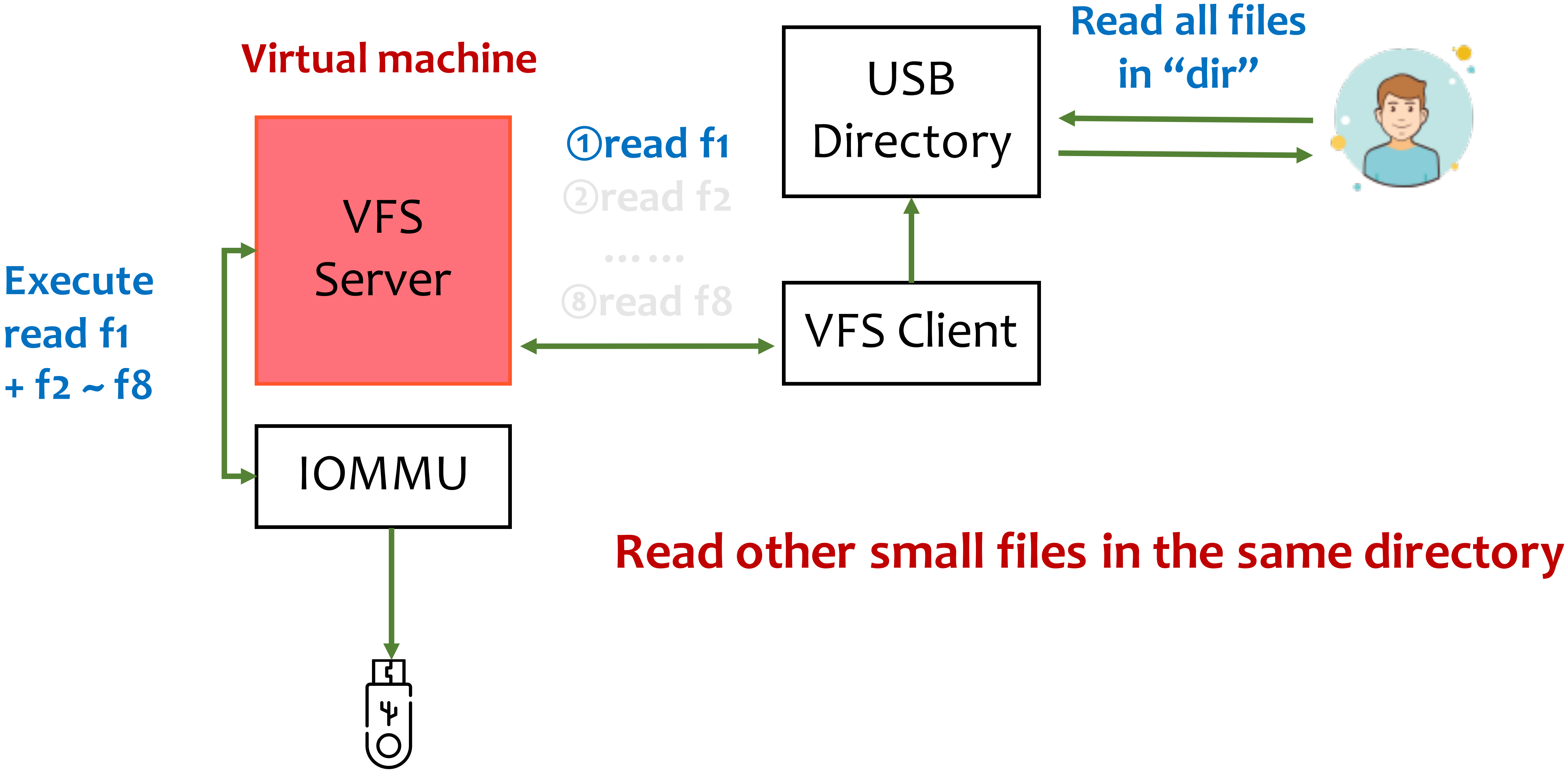
①read f1
②read f2
……
⑧read f8

USB Directory

VFS Client

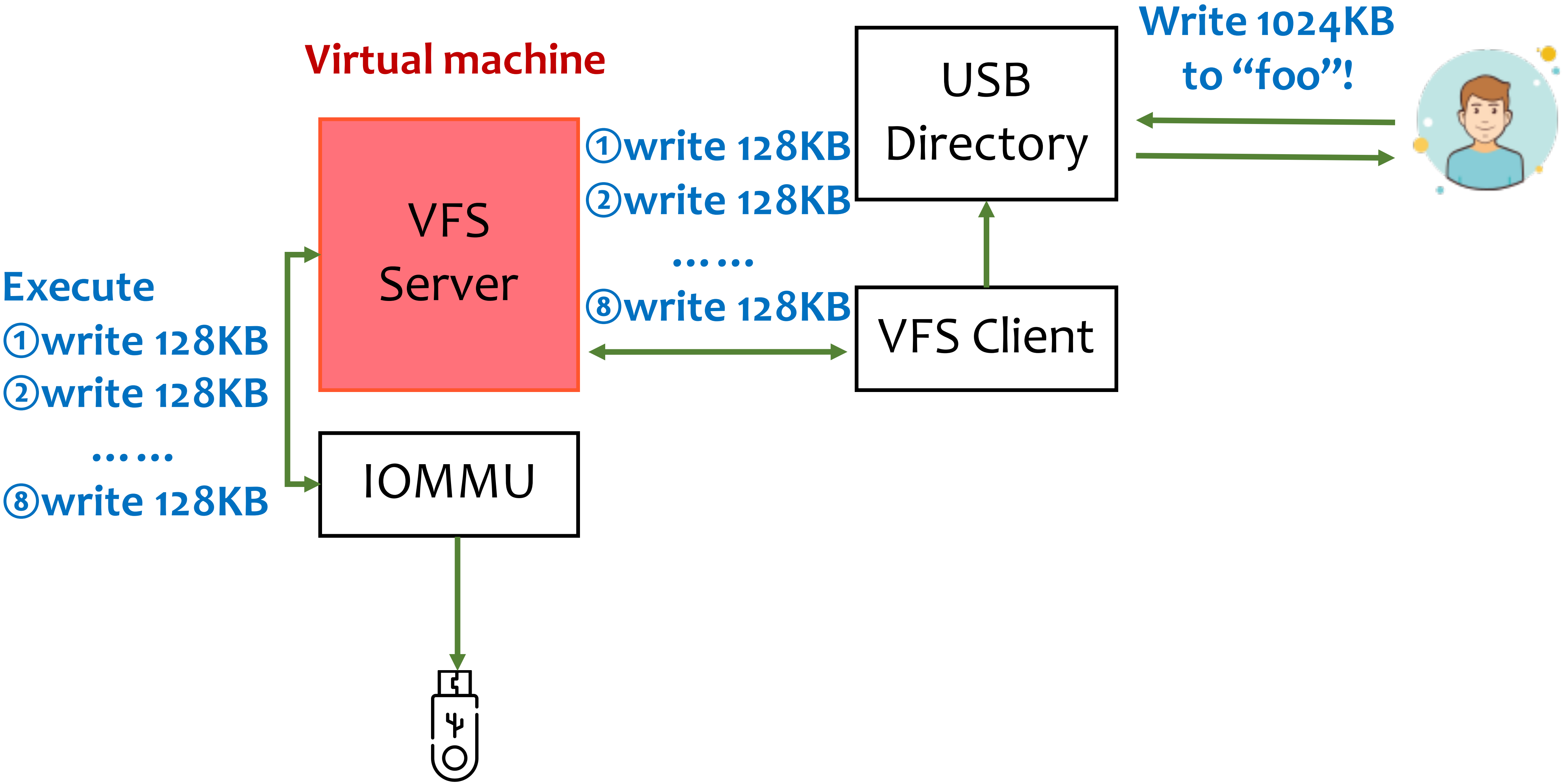
Read all files in "dir"

Prefetching



Read other small files in the same directory

Batching operations



Virtual machine

VFS Server

IOMMU

Execute
①write 128KB
②write 128KB
……
⑧write 128KB

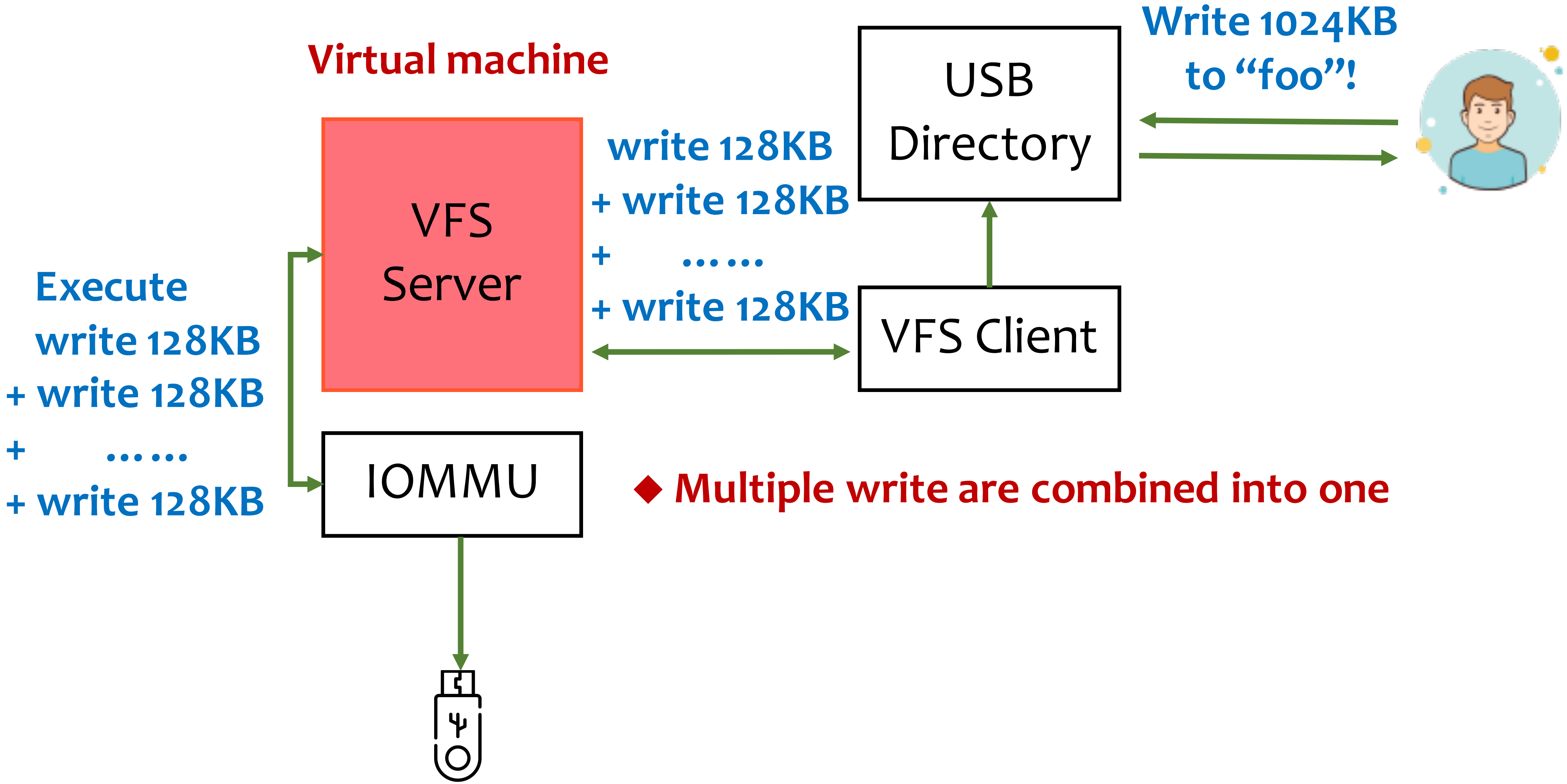
①write 128KB
②write 128KB
……
⑧write 128KB

USB Directory

VFS Client

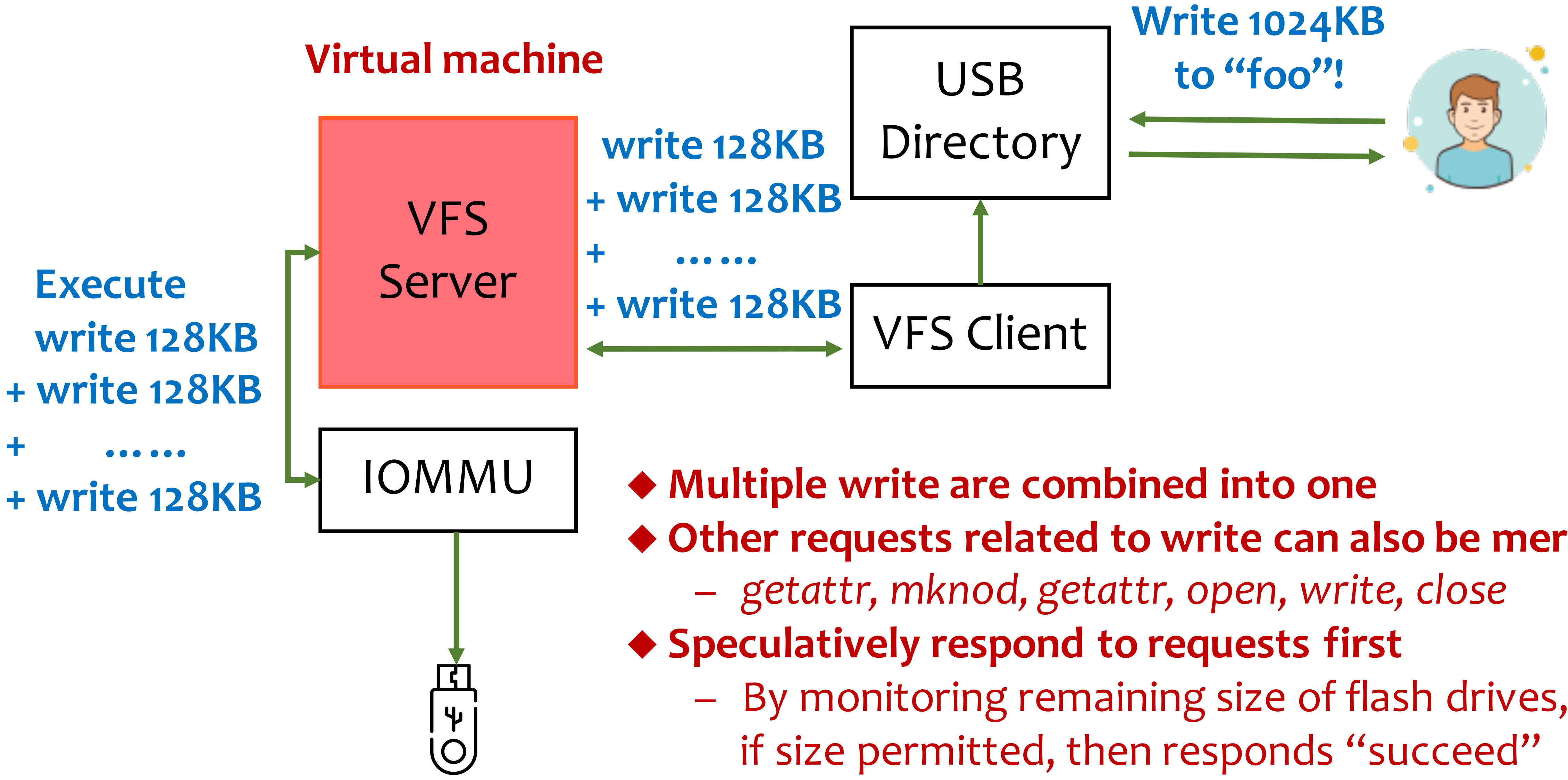
Write 1024KB to “foo”!

Batching operations



◆ Multiple write are combined into one

Batching operations

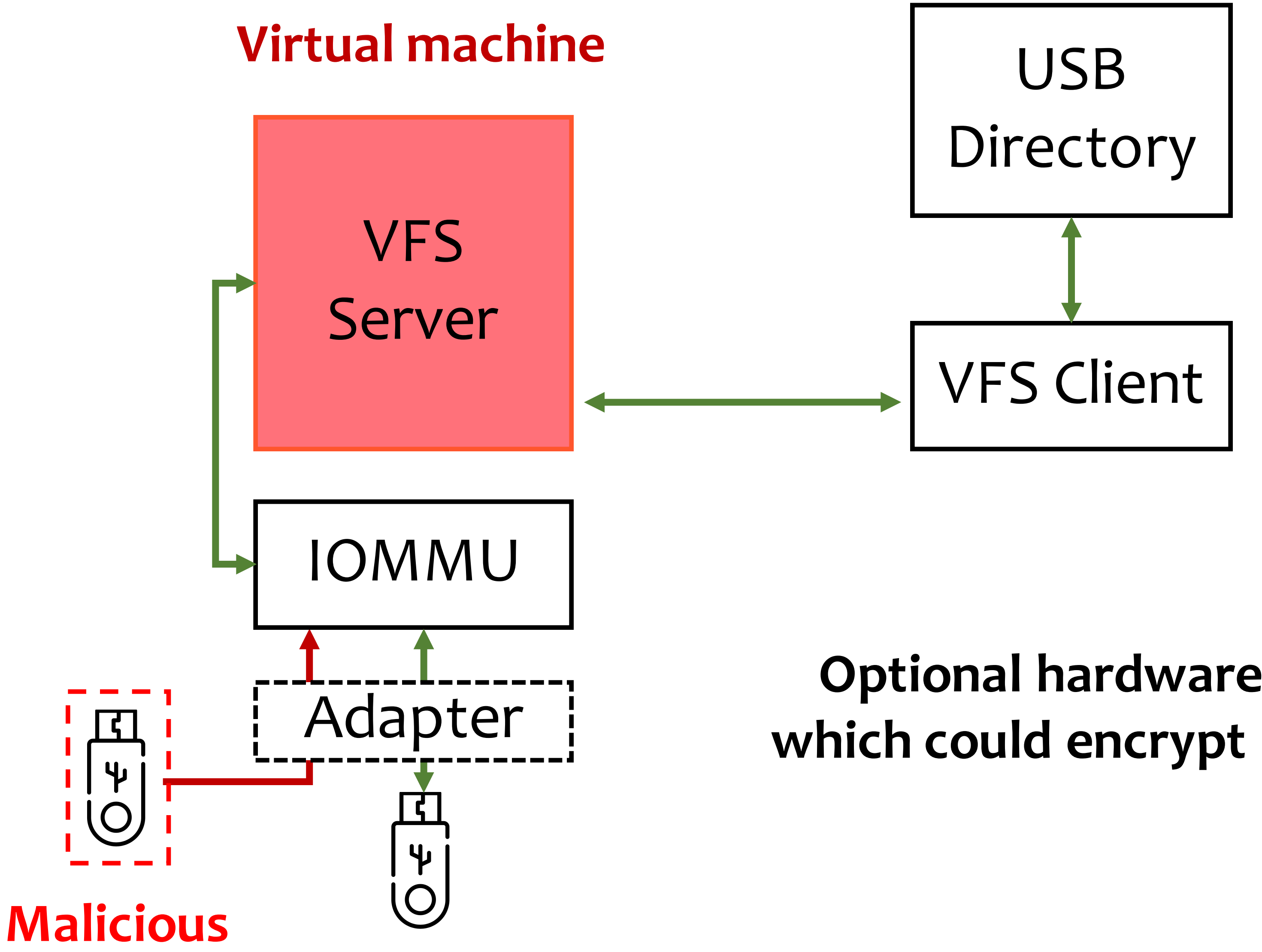


- ◆ **Multiple write are combined into one**
- ◆ **Other requests related to write can also be merged**
  - – *getattr, mknod, getattr, open, write, close*
- ◆ **Speculatively respond to requests first**
  - – By monitoring remaining size of flash drives, if size permitted, then responds "succeed"

Agenda

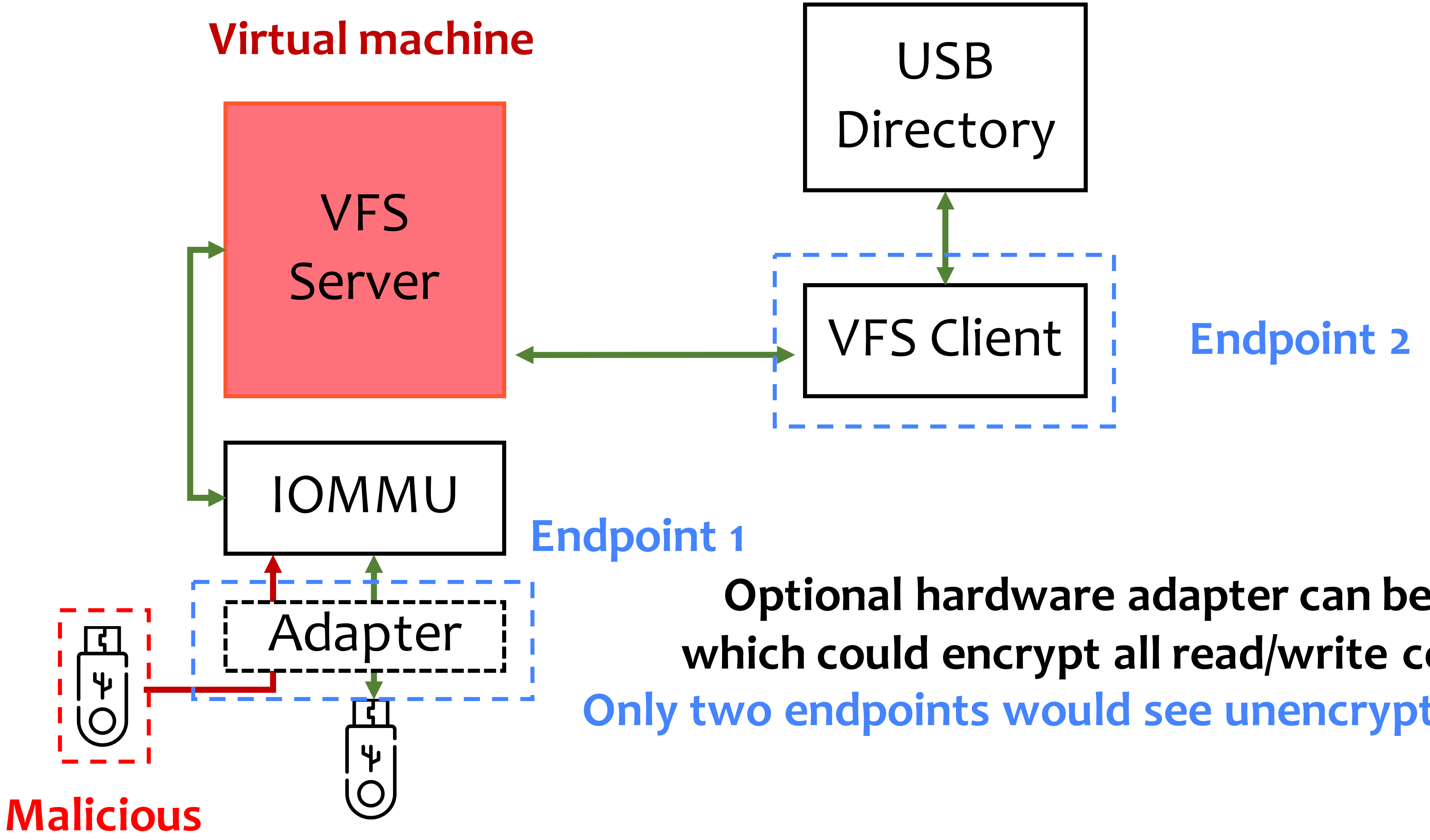
- **How to address those challenges**
  - Optimizations
  - Encrypted communication
  - Formally verified serializer and parser
- **Preliminary evaluation**
- **Conclusion & Discussion**

Encrypted communication



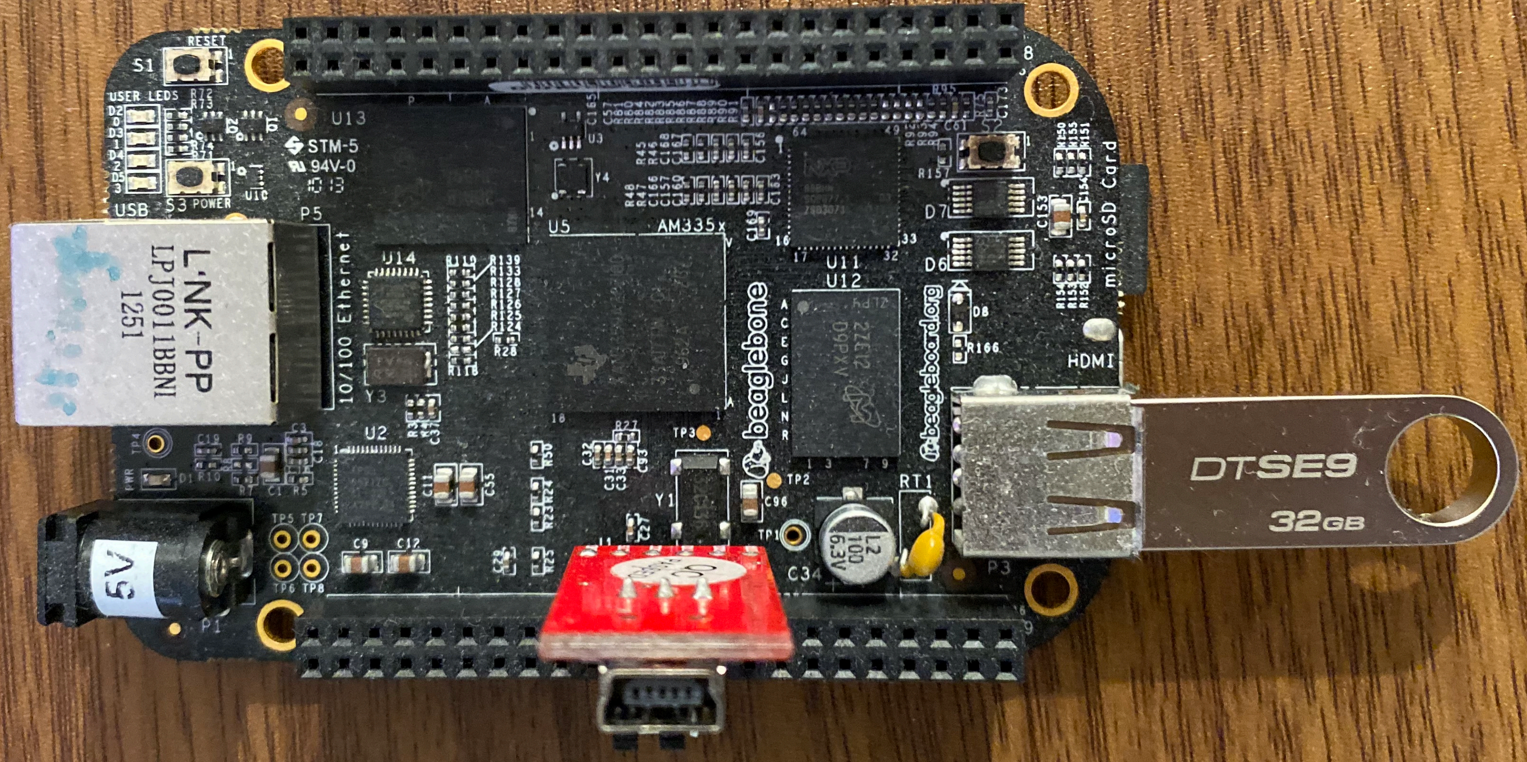
Optional hardware adapter can be used which could encrypt all read/write contents

Encrypted communication



Optional hardware adapter can be used which could encrypt all read/write contents

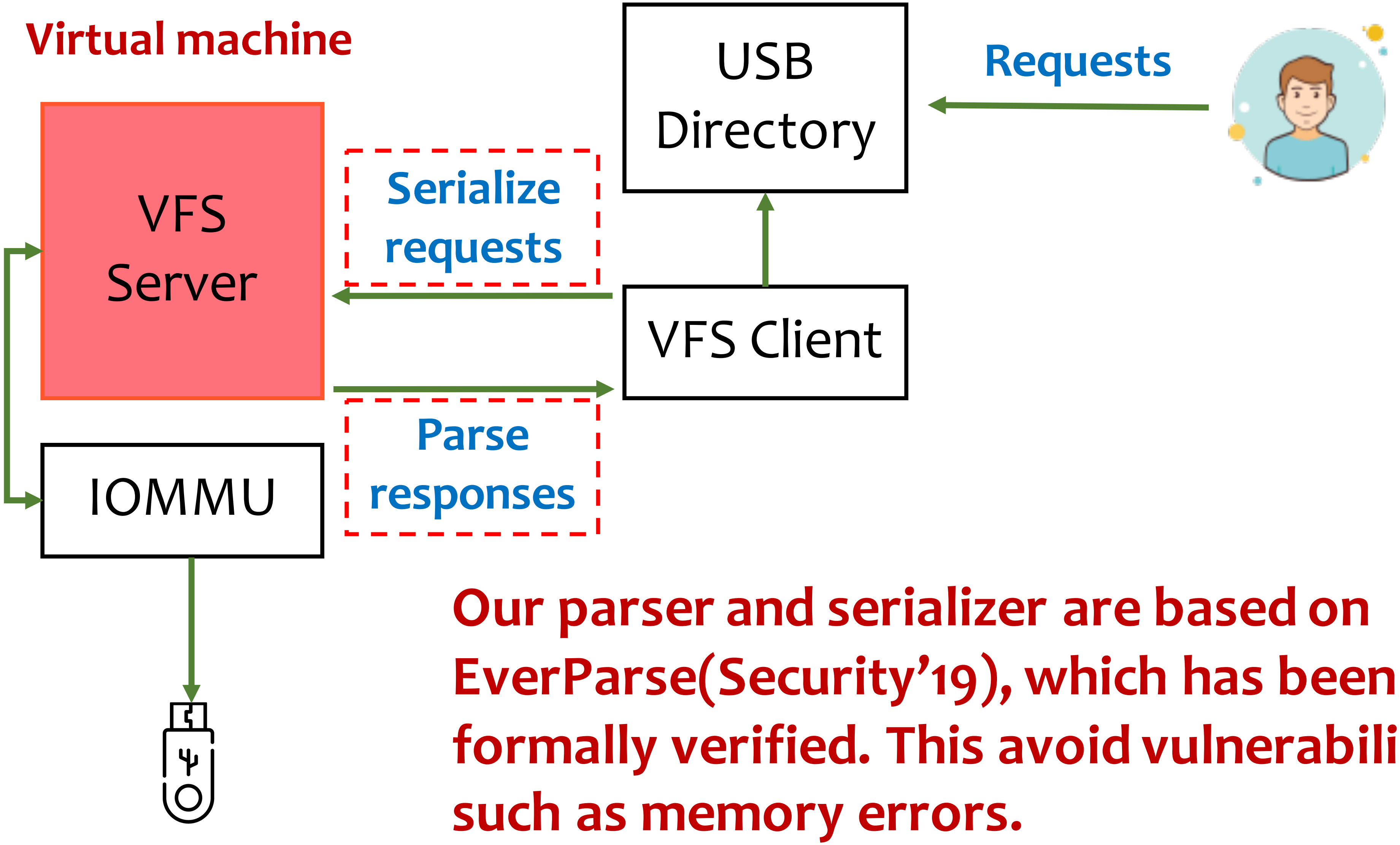
Only two endpoints would see unencrypted contents


RESET
S1
USER LEDS
USB
S3
POWER
U13
STM-5
94V-0
P5
L-NK-PP
LPJ0011BBNL
1251
10/100 Ethernet
U14
Y3
U2
U5
AM335x
U11
U12
beaglebone
beagleboard.org
microSD Card
HDMI
D7
D6
S2
Y1
TP1
TP2
TP3
C34
RT1
P3
P1
5V
DTSE9
32GB

Agenda

- ◆ **How to address those challenges**
  - Optimizations
  - Encrypted communication
  - Formally verified serializer and parser
- ◆ **Preliminary evaluation**
- ◆ **Conclusion & Discussion**

Formally verified serializer and parser



Our parser and serializer are based on EverParse(Security'19), which has been formally verified. This avoid vulnerabilities, such as memory errors.

Agenda

- How to address those challenges
  - Optimizations
  - Encrypted communication
  - Formally verified serializer and parser

- **Preliminary evaluation**

- Discussion & Conclusion

Experiment setup

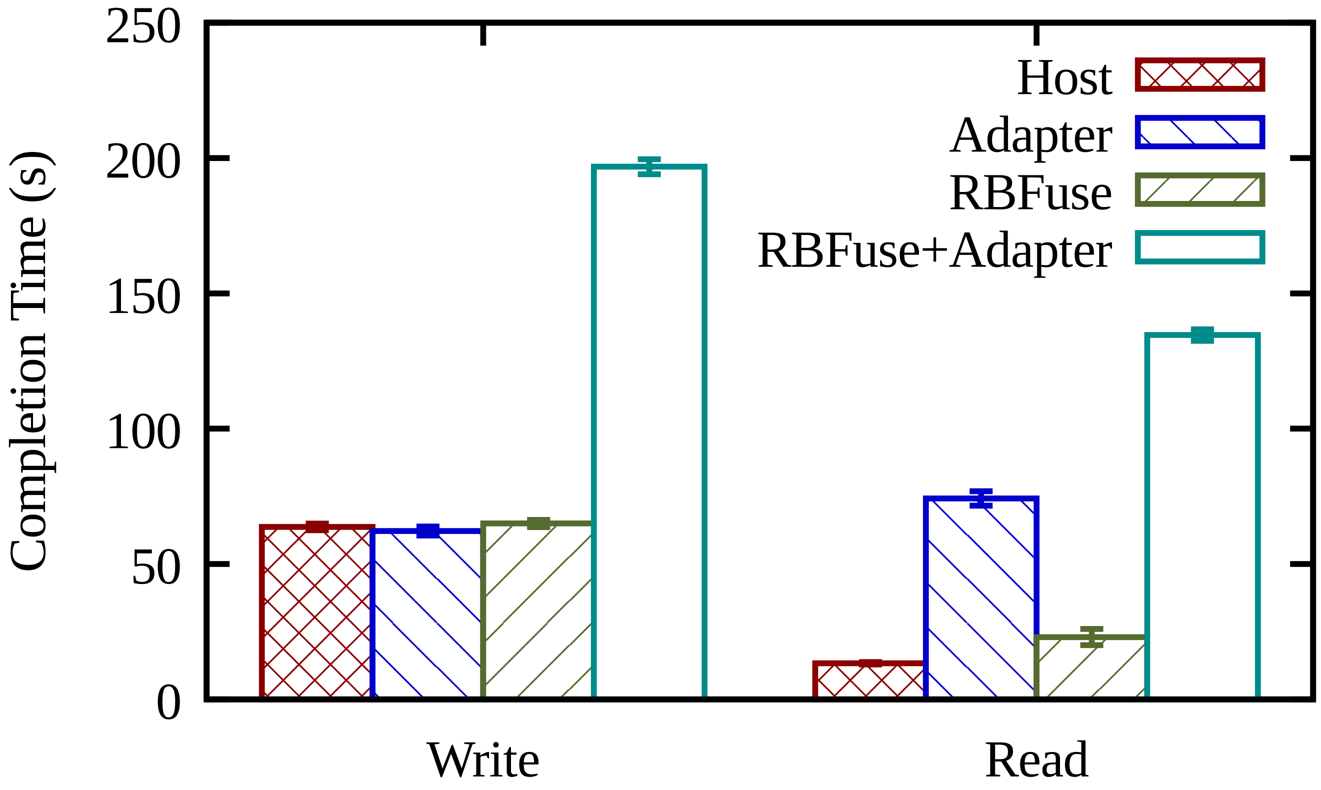
- ◆ For virtual machine we run Ubuntu 16.04 (Linux 4.15.0-45) on QEMU.
  Host machine is also Ubuntu 16.04 with KVM.

- ◆ Adapter for authentication and data encryption is built on a BeagleBone Black which runs Debian 9.1 (Linux 4.4.88-ti-r125).

- ◆ We used *filebench* to run our experiments.

- ◆ Our baseline is flash drive connected to the host without any of our mechanisms.

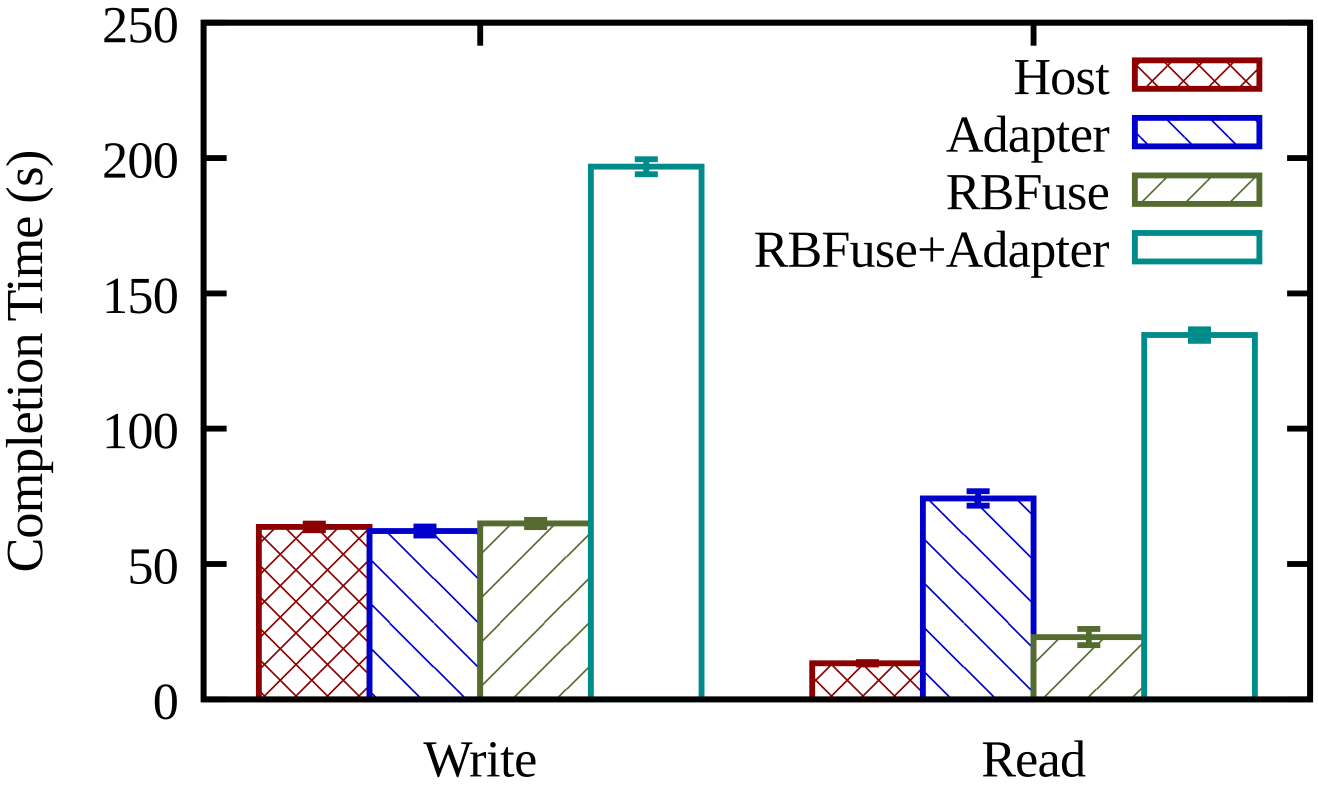
One large file (500MB)



Takeaway:

① *RBFuse* itself brings little overhead

One large file (500MB)

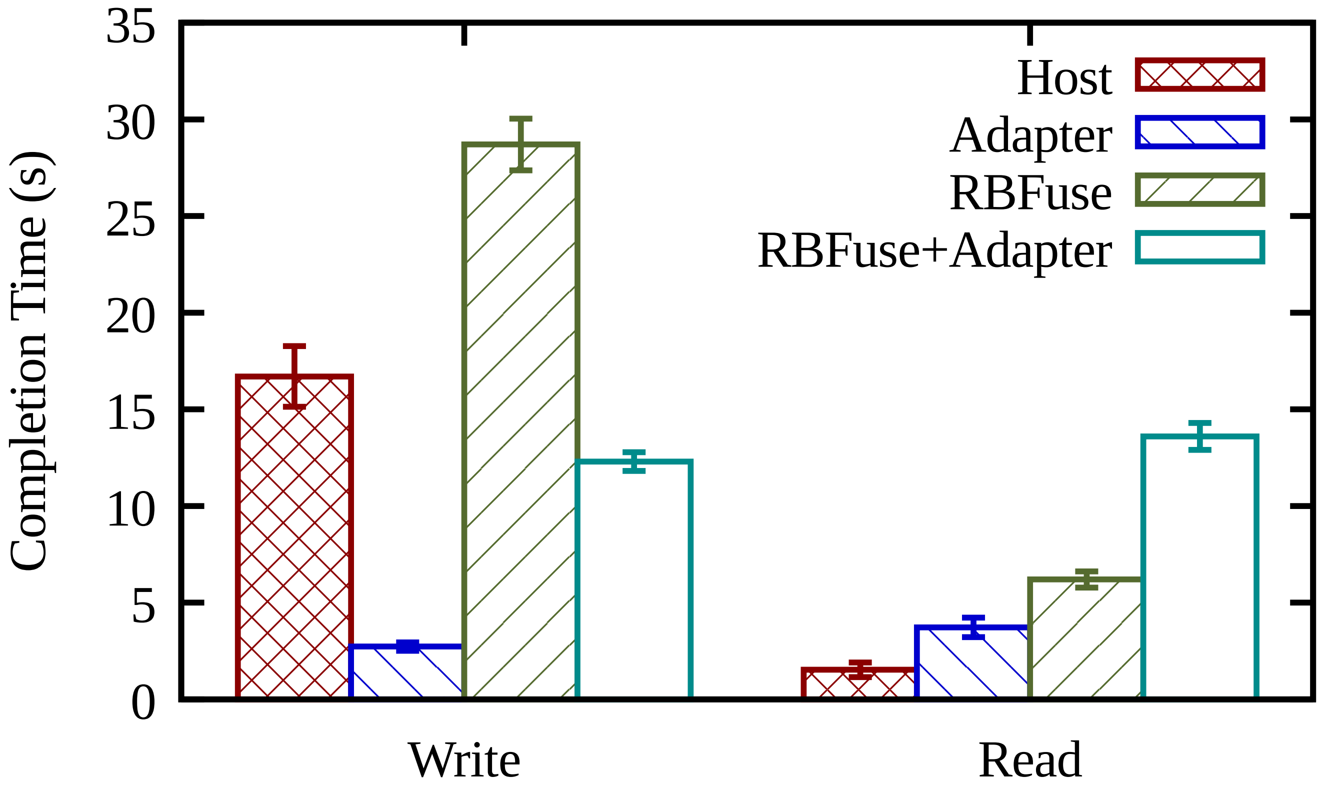


Takeaway:

① *RBFuse* itself brings little overhead

② *RBFuse* + adapter brings about 3x-10x overhead, due to the bad performance of adapter and increased roundtrips between flash drive and host

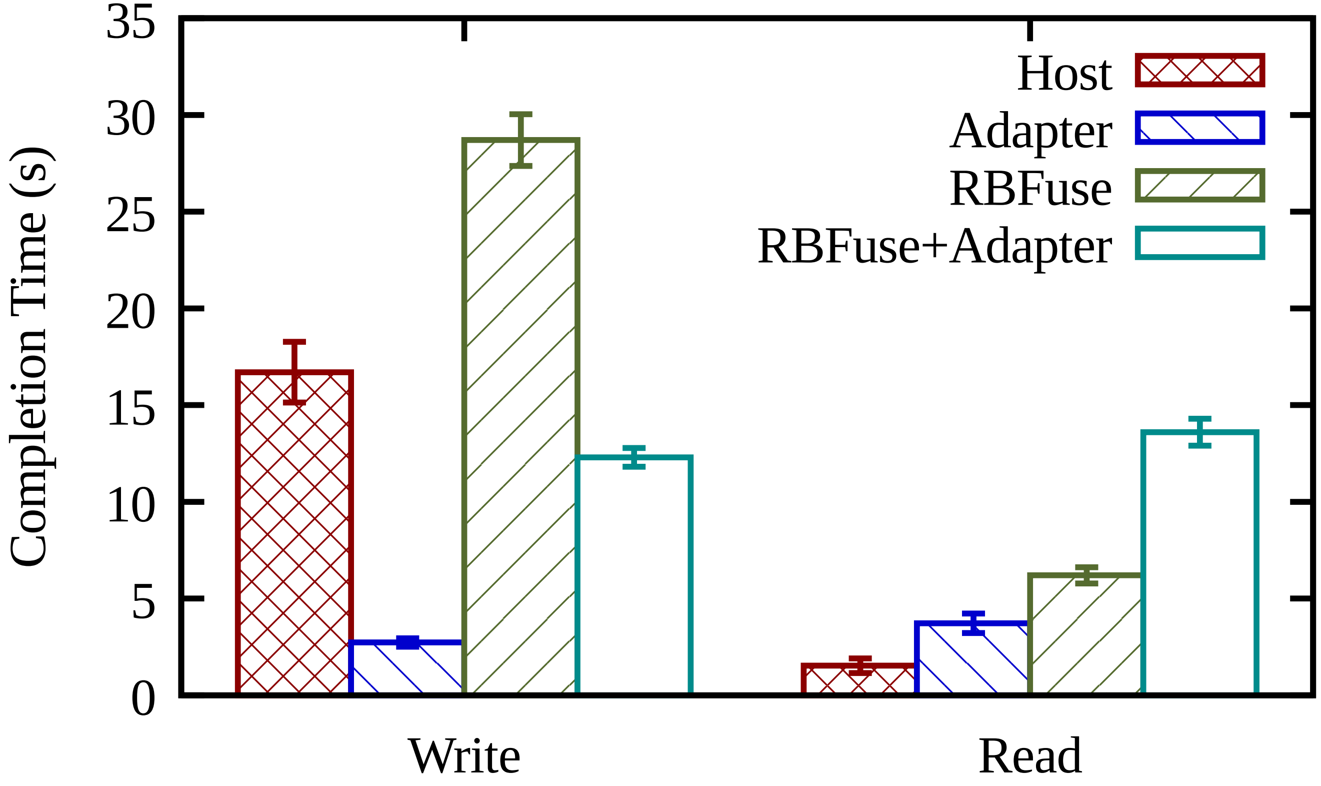
1,000 small files (16KB each)



Takeaway:

① *RBFuse* itself brings 2x-4x overhead

1,000 small files (16KB each)



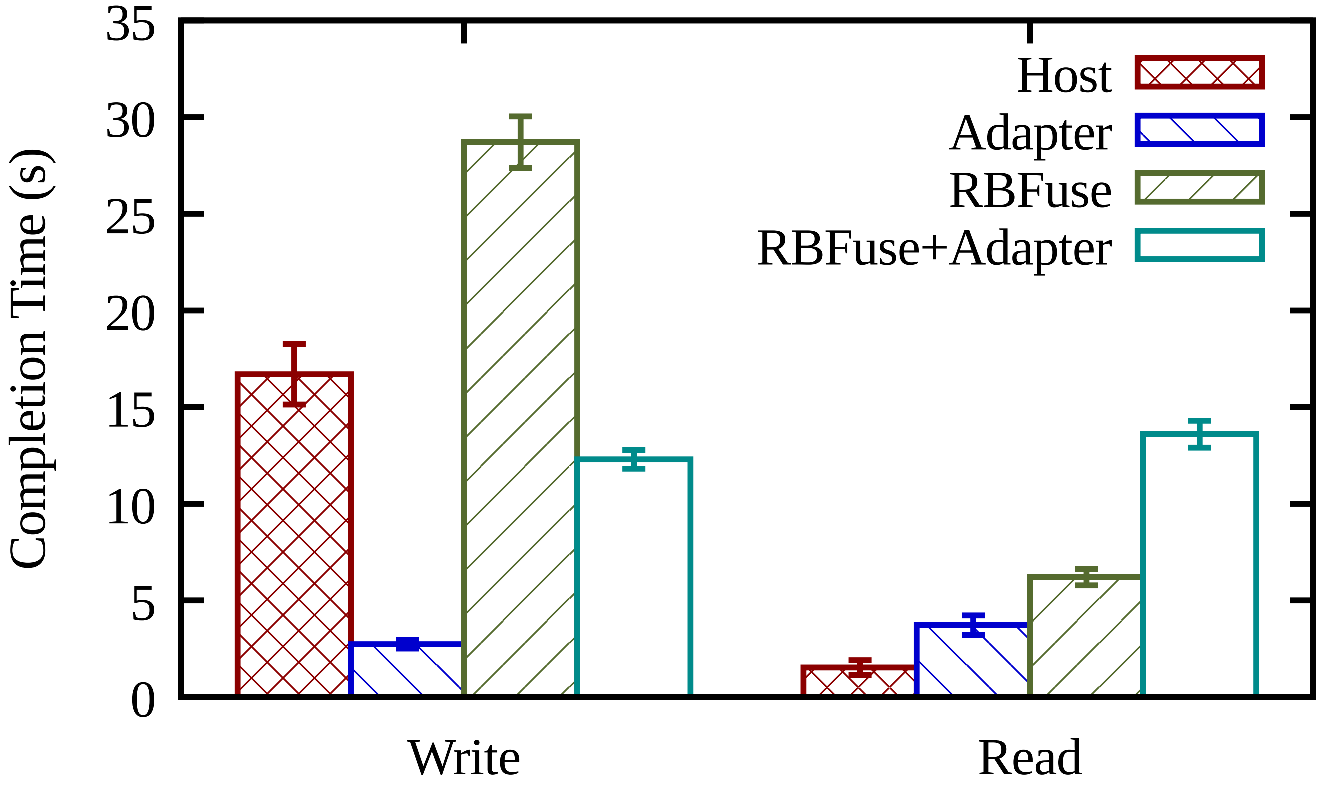
Takeaway:

① *RBFuse* itself brings 2x-4x overhead

② For *write*, *RBFuse* + adapter outperforms directly accessing due to better performance of adapter on this task

1,000 small files (16KB each)

Note: adapter could be viewed as another machine with Debian



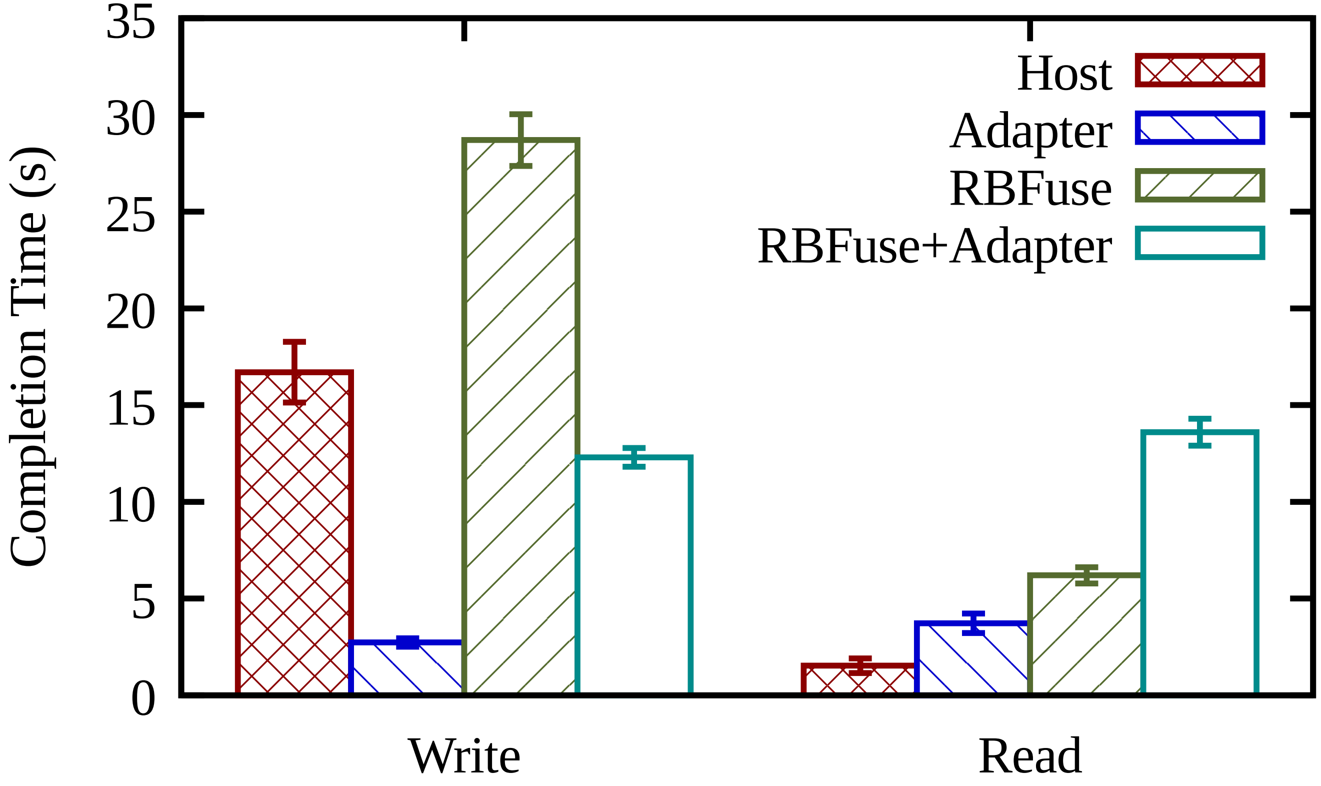
Takeaway:

① *RBFuse* itself brings 2x-4x overhead

② For *write*, *RBFuse* + adapter outperforms directly accessing due to better performance of adapter on this task

1,000 small files (16KB each)

Note: adapter could be viewed as another machine with Debian



Takeaway:

① *RBFuse* itself brings 2x-4x overhead

② For *write*, *RBFuse* + adapter outperforms directly accessing due to better performance of adapter on this task

③ For read, *RBFuse* + adapter brings 8.8x overhead, due to that adapter itself would bring about 2x overhead

Agenda

- How to address those challenges
  - Optimizations
  - Encrypted communication
  - Formally verified serializer and parser

- Preliminary evaluation

- **Discussion & Conclusion**

Crash consistency test

- **We modified and ran crashmonkey (OSDI'18) on *RBFuse***
  - ext4
  - vfat

Crash consistency test

- **We modified and ran crashmonkey (OSDI'18) on *RBFuse***
  - ext4
  - vfat

```
open foo O_RDWR|O_CREAT 0777
fsync foo
checkpoint 1
close foo
```

Crash consistency test

- **We modified and ran crashmonkey (OSDI'18) on *RBFuse***
  - ext4
  - vfat

```
open foo O_RDWR|O_CREAT 0777
fsync foo
checkpoint 1
close foo
```

vfat and RBFuse on vfat would fail!

Crash consistency test

- **We modified and ran crashmonkey (OSDI'18) on *RBFuse***
  - ext4
  - vfat

```
open foo O_RDWR|O_CREAT 0777
fsync foo
checkpoint 1
close foo
```

Hydra (SOSP'19):
Parent directory of foo
need to be ***sync***

vfat and RBFuse on vfat would fail!

Previous file system fuzzing

- Janus (S&P’19): two-dimensional input fuzzing

File system

User(Client)

Previous file system fuzzing

- ◆ **Janus (S&P’19): two-dimensional input fuzzing**

Image fuzzing

File system

User(Client)

Previous file system fuzzing

- Janus (S&P'19): two-dimensional input fuzzing

Image fuzzing

File system

Client program fuzzing

User(Client)

Server side fuzzing

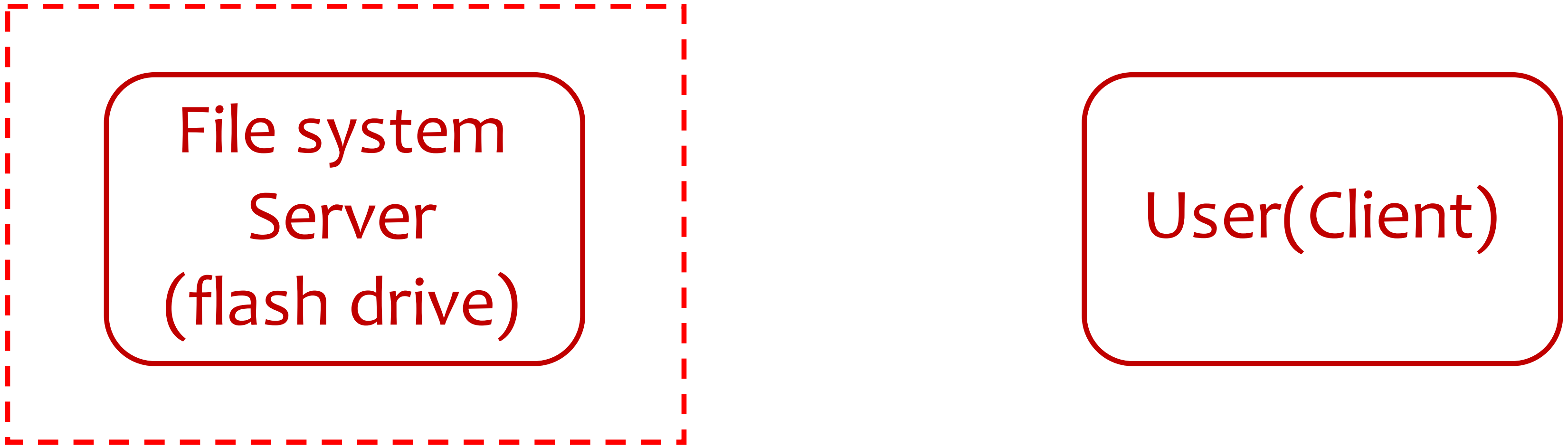
- We assume the (file system) server is malicious

File system
Server
(flash drive)

User(Client)

Server side fuzzing

- We assume the (file system) server is malicious



Malicious messages fuzzing

Formal verification

- **VFS interface is small, has better-defined semantics than USB**

- **Formal verification on our system**
  - Getting the virtual file system interface “right”

Conclusion

- **We propose *RBFuse*, which is a file system that accesses flash drives without interacting with the USB stack on the host machine with reasonable overhead**

- **Discussion**
  - Crash consistency test for *RBFuse*
  - Server side fuzzing
  - Formal verification

Thank you! Any questions or suggestions?

- **We propose *RBFuse*, which is a file system that accesses flash drives without interacting with the USB stack on the host machine with reasonable overhead**

- **Discussion**
  - Crash consistency test for *RBFuse*
  - Server side fuzzing
  - Formal verification