LIBMPK: SOFTWARE ABSTRACTION FOR INTEL MEMORY PROTECTION KEYS (INTEL MPK)

Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon and Taesoo Kim

SECURITY CRITICAL MEMORY REGIONS NEED PROTECTION

- JIT page

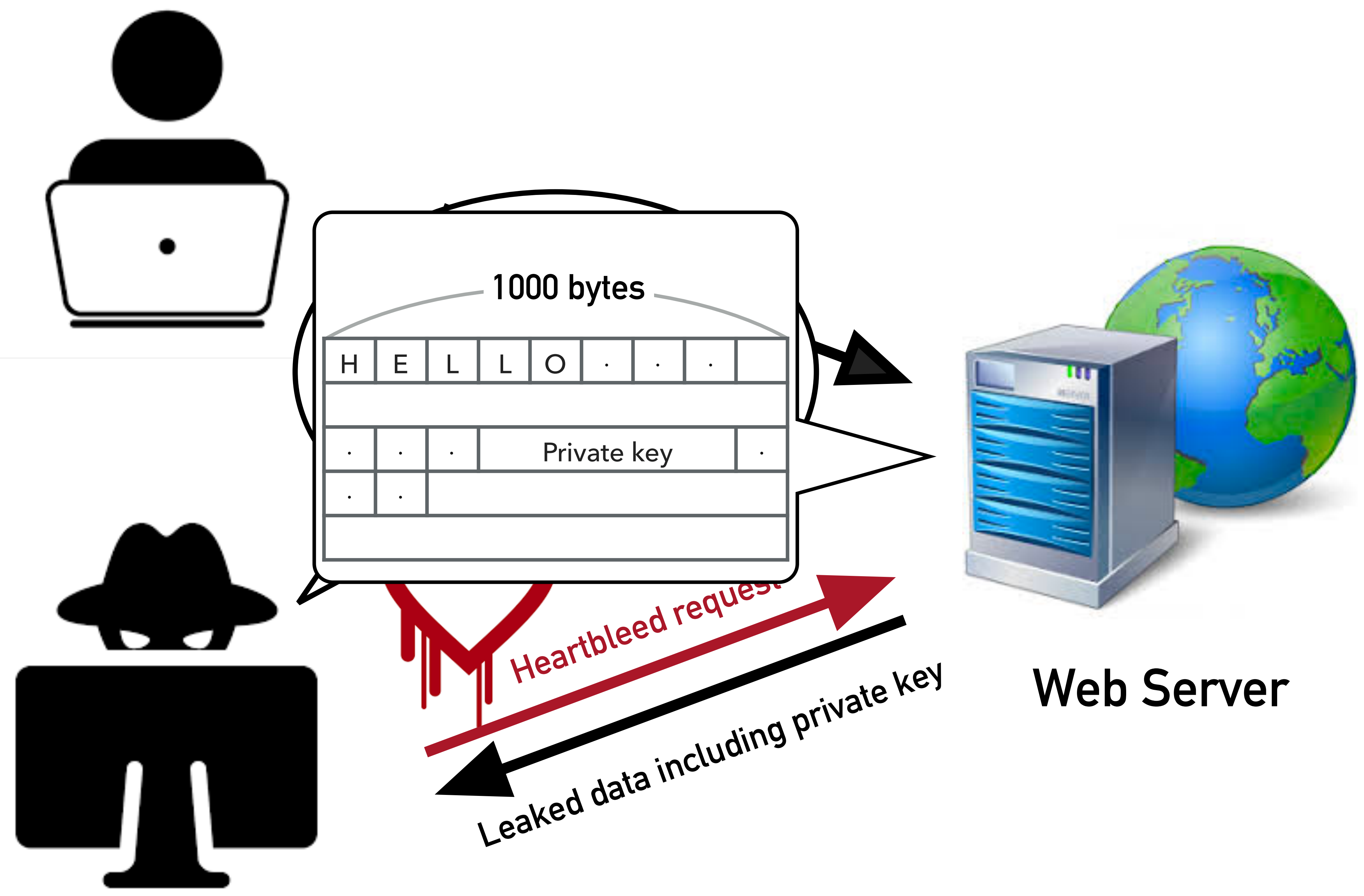
  "To achieve code execution, we can simply locate one of these RWX JIT pages and overwrite it with our own shellcode." - [1]

- Personal information
- Password
- Private key

  "We confirmed that all individuals used only the Heartbleed exploit to obtain the private key." - [2]

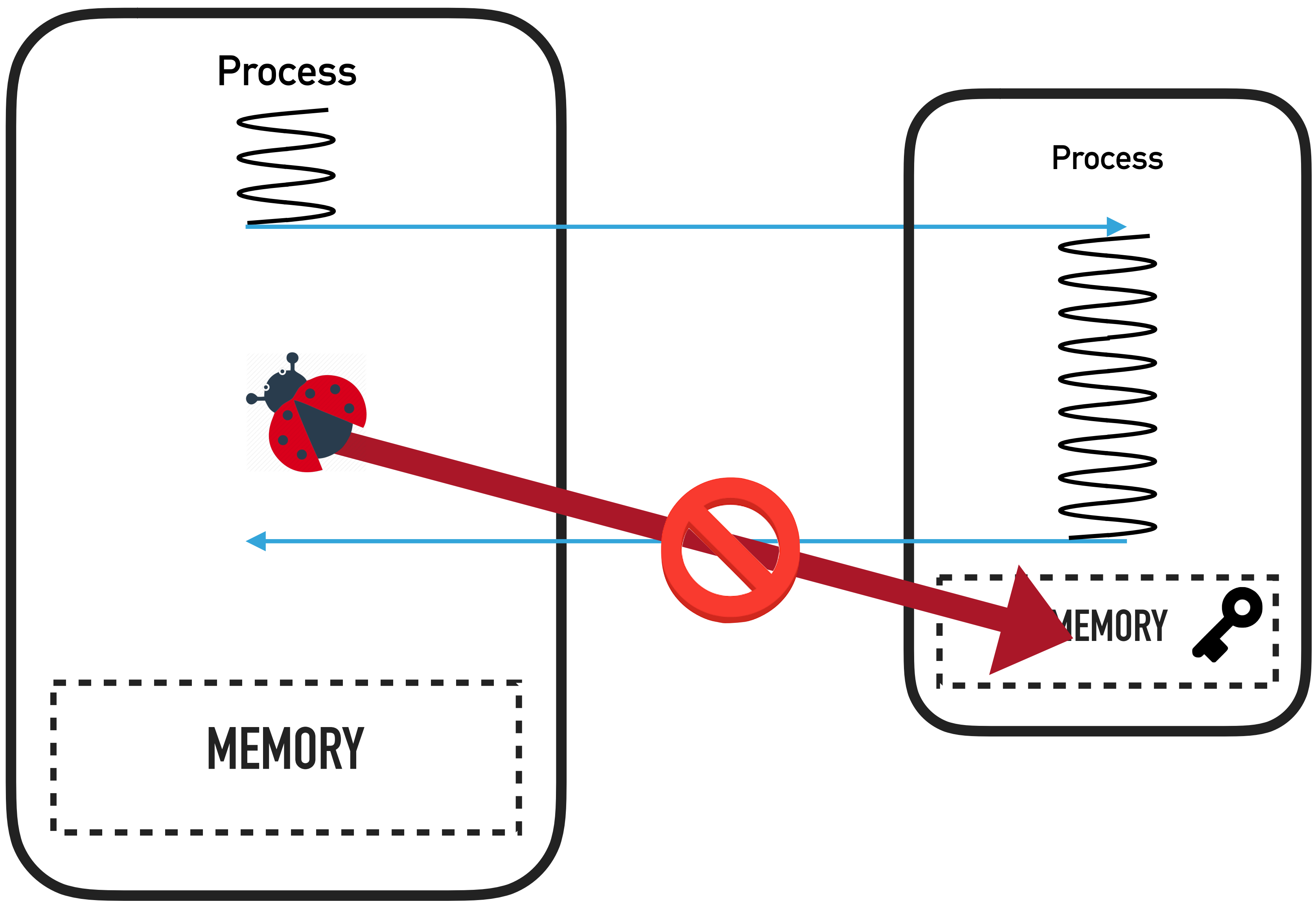
[1] Amy Burnett, et al. "Weaponization of a Javascriptcore vulnerability" RET2 Systems Engineering Blog
[2] Nick Sullivan "The Results of the CloudFlare Challenge" CloudFlare Blog

EXAMPLE 1 – HEARTBLEED ATTACK



EXAMPLE 1 : EXISTING SOLUTION TO PROTECT MEMORY

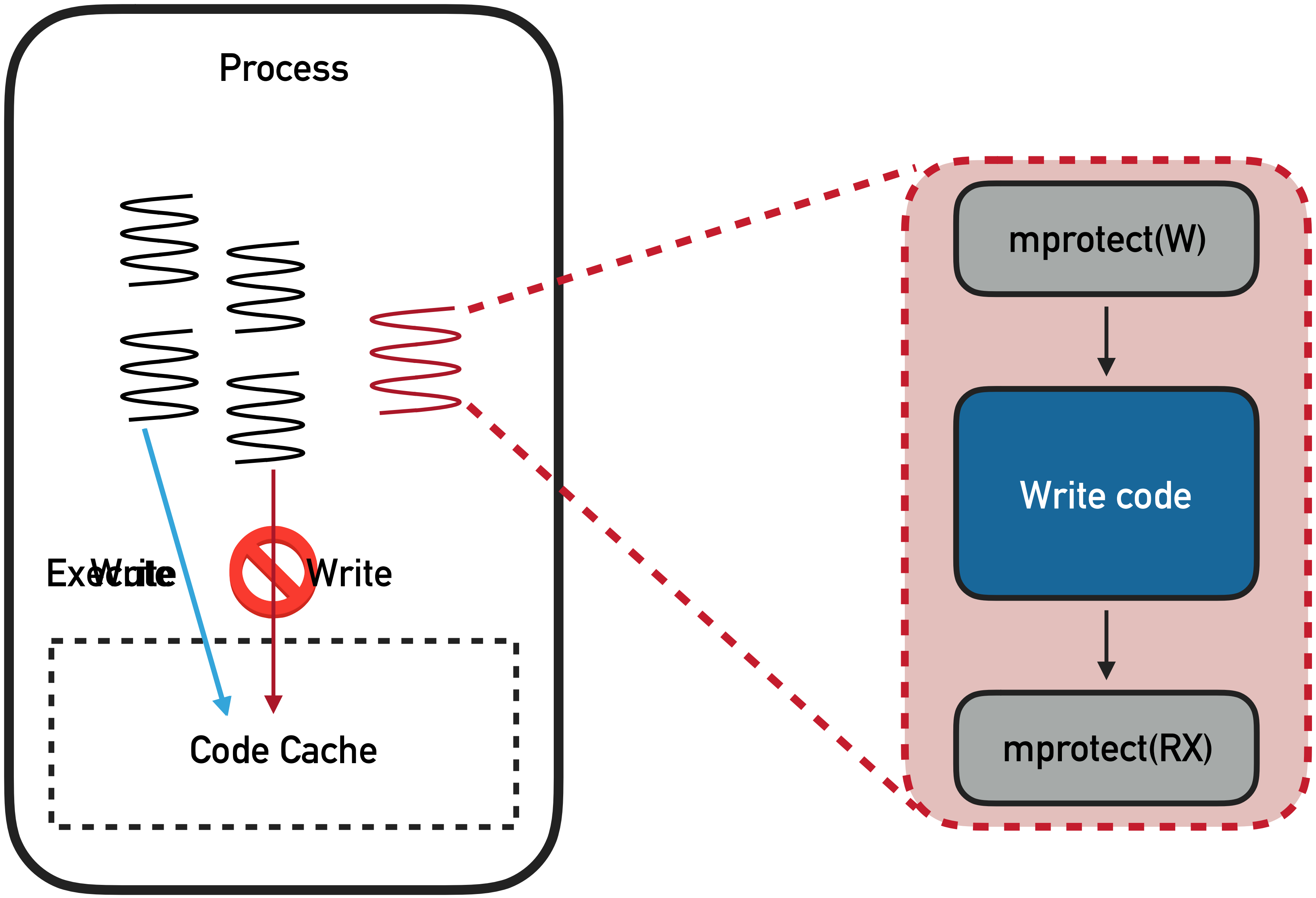
- Process separation



[1] Song, Chengyu, et al. "Exploiting and Protecting Dynamic Code Generation", NDSS 2015.
[2] Litton, James, et al. "Light-Weight Contexts: An OS Abstraction for Safety and Performance", OSDI 2016.

EXAMPLE 2 – EXISTING SOLUTION TO PROTECT JIT PAGE

- JIT page W^X protection



PROBLEMS OF EXISTING SOLUTIONS

- Process Separation

High overhead to spawn new process and synch data

- W^X Protection

Multiple cost to change permission of multiple pages

Race condition due to permission synchronization

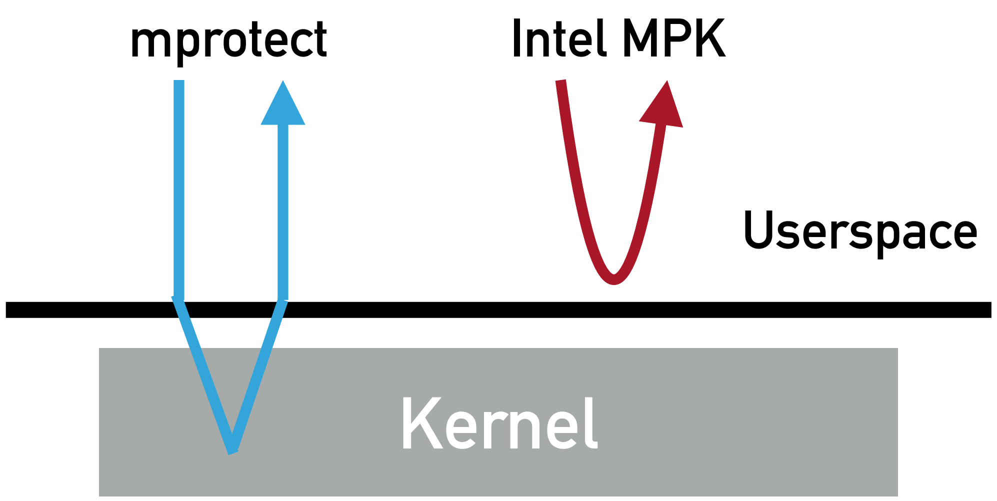
This talk: utilizing a hardware mechanism, Intel Memory Protection Key (MPK), to address these challenges

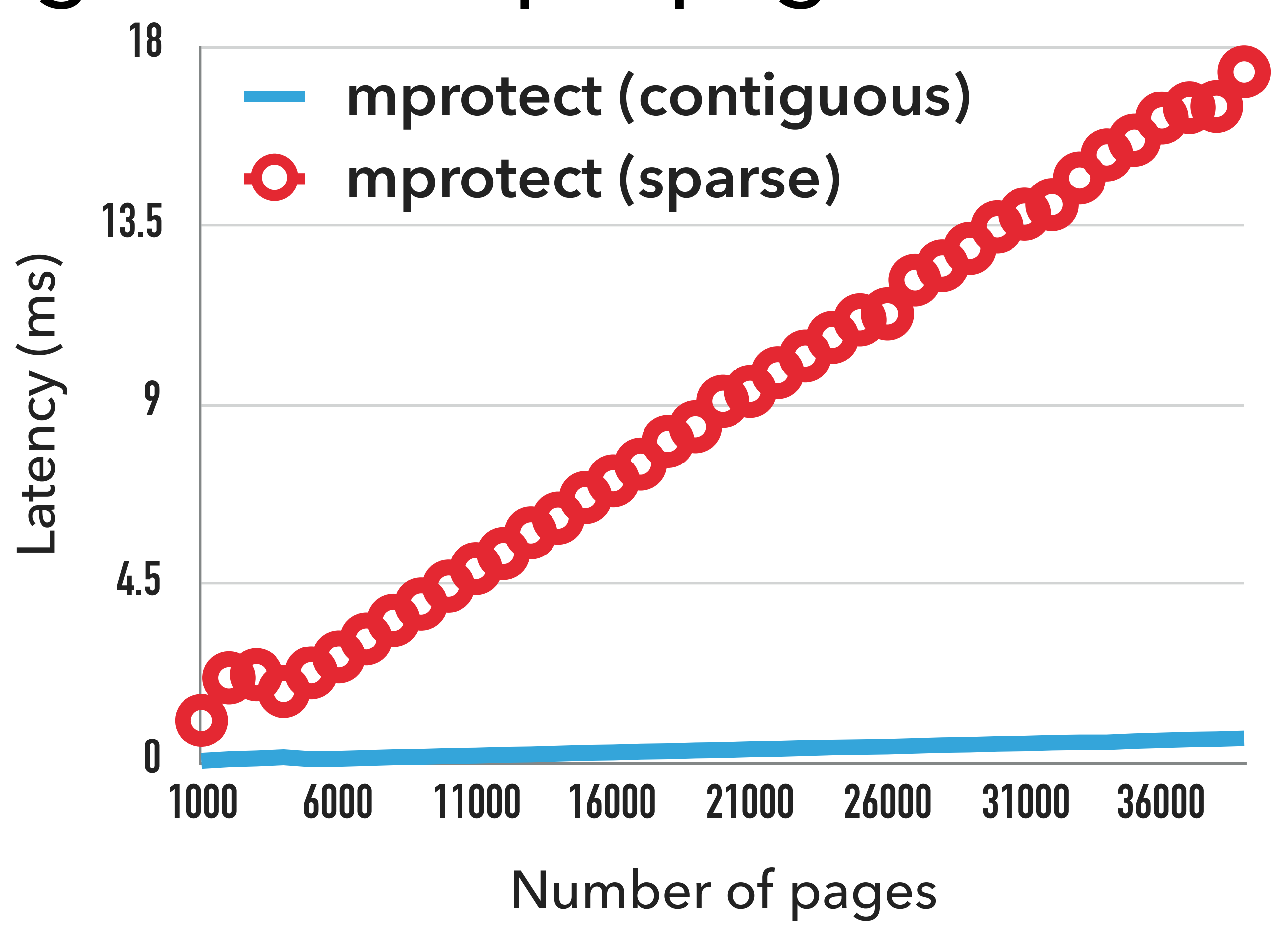
OUTLINE

- Introduction
- **Intel MPK Explained**
- Challenges
- Design
- Implementation
- Evaluation
- Discussion
- Related Work
- Conclusion

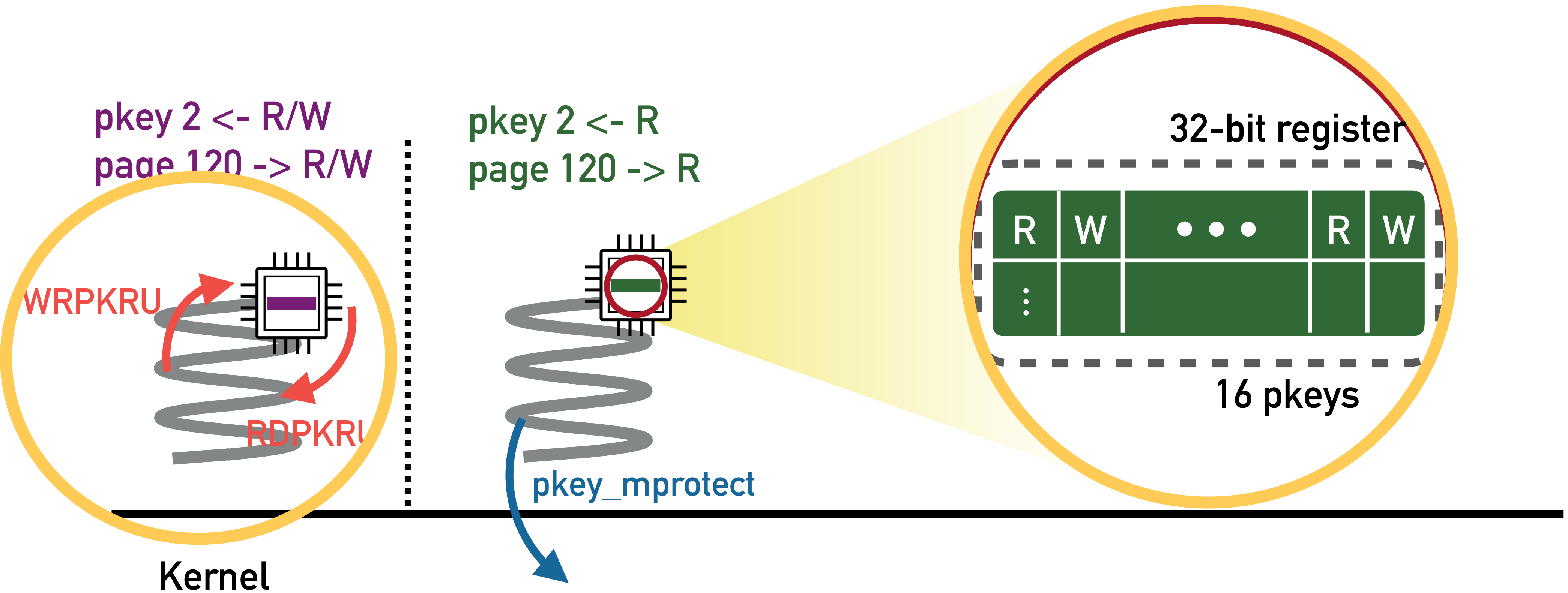
OVERVIEW

- Support fast permission change for page groups with single instruction
  - Fast single invocation
  - Fast permission change for multiple pages





UNDERLINE IMPLEMENTATION



page #	pkey	· · ·	perm.
120	2	· · ·	R/W
· · ·			

< Page table>

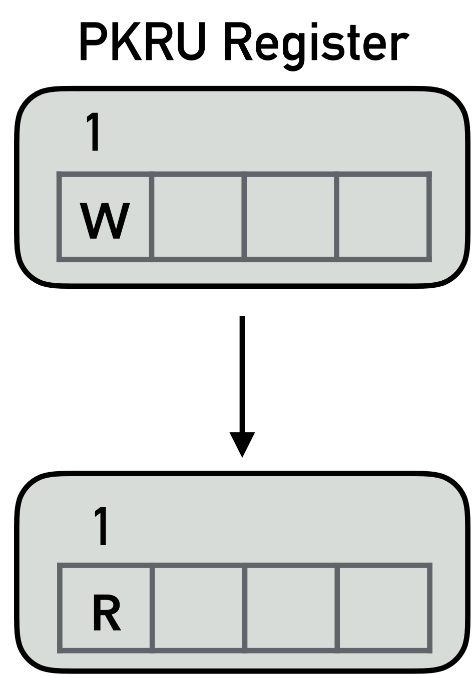
- Permissions per cpu
- 32-bit PKRU register contains keys/perm
  - WRPKRU: write key/perm
  - RDPKRU: read key/perm

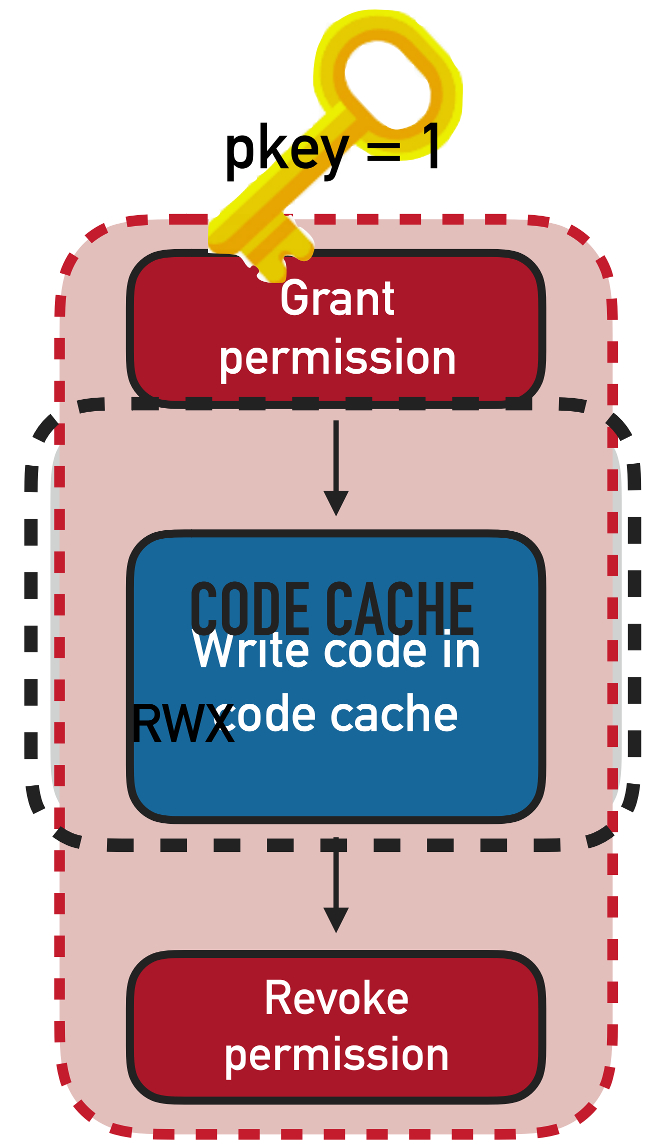
EXAMPLE – JIT PAGE W^X PROTECTION

```
function init()
   pkey = pkey_alloc()
   pkey_mprotect(code_cache, len, RWX, pkey)

function JIT()
   WRPKRU(pkey, W)
   ...
   write code cache
   ...
   WRPKRU(pkey, R)

function fini()
   pkey_free(pkey)
```



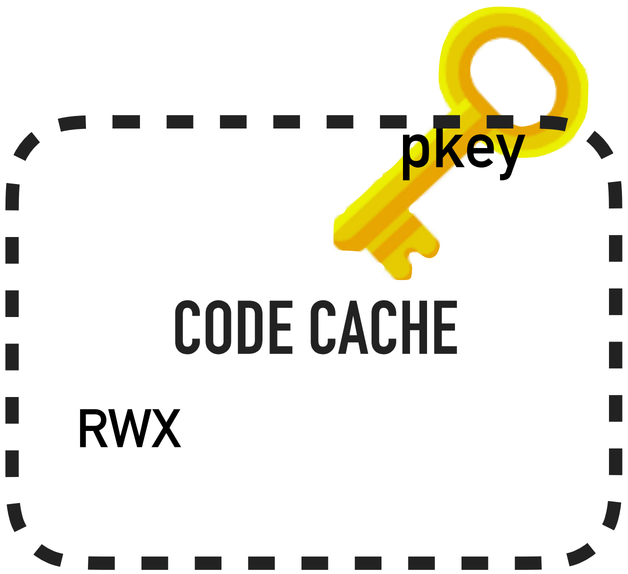


EXAMPLE : EXECUTABLE-ONLY MEMORY

```
function init()
  pkey = pkey_alloc()
  pkey_mprotect(code_cache, len, RWX, pkey)

function JIT()
  WRPKRU(pkey, W)
  ...
  write code cache
  ...
  WRPKRU(pkey, R)

function fini()
  pkey_free(pkey)
```

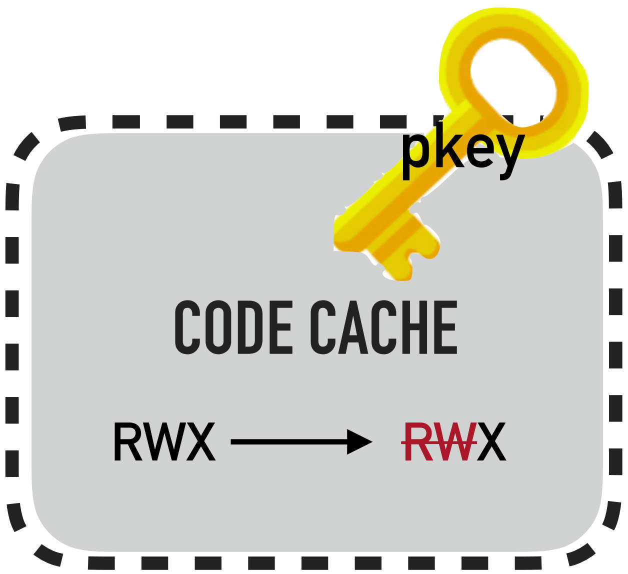


EXAMPLE : EXECUTABLE-ONLY MEMORY

```
function init()
  pkey = pkey_alloc()
  pkey_mprotect(code_cache, len, RWX, pkey)

function JIT()
  WRPKRU(pkey, W)
  ...
  write code cache
  ...
  WRPKRU(pkey, None)

function fini()
  pkey_free(pkey)
```

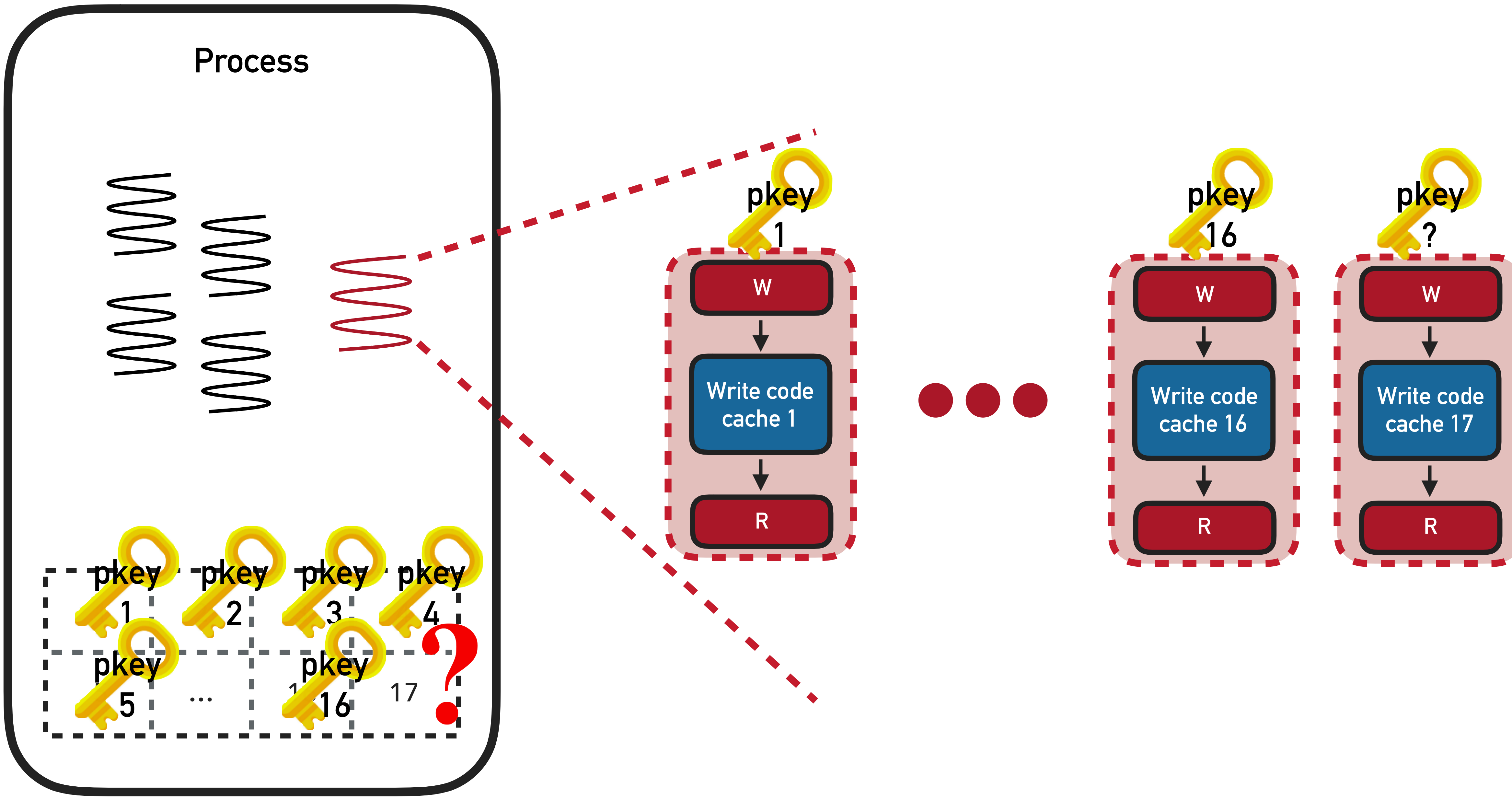


OUTLINE

- Introduction
- Intel MPK Explained
- **Challenges**
  - **Non-scalable Hardware Resource**
  - **Asynchronous Permission Change**
- Design
- Implementation
- Evaluation
- Discussion
- Related Work
- Conclusion

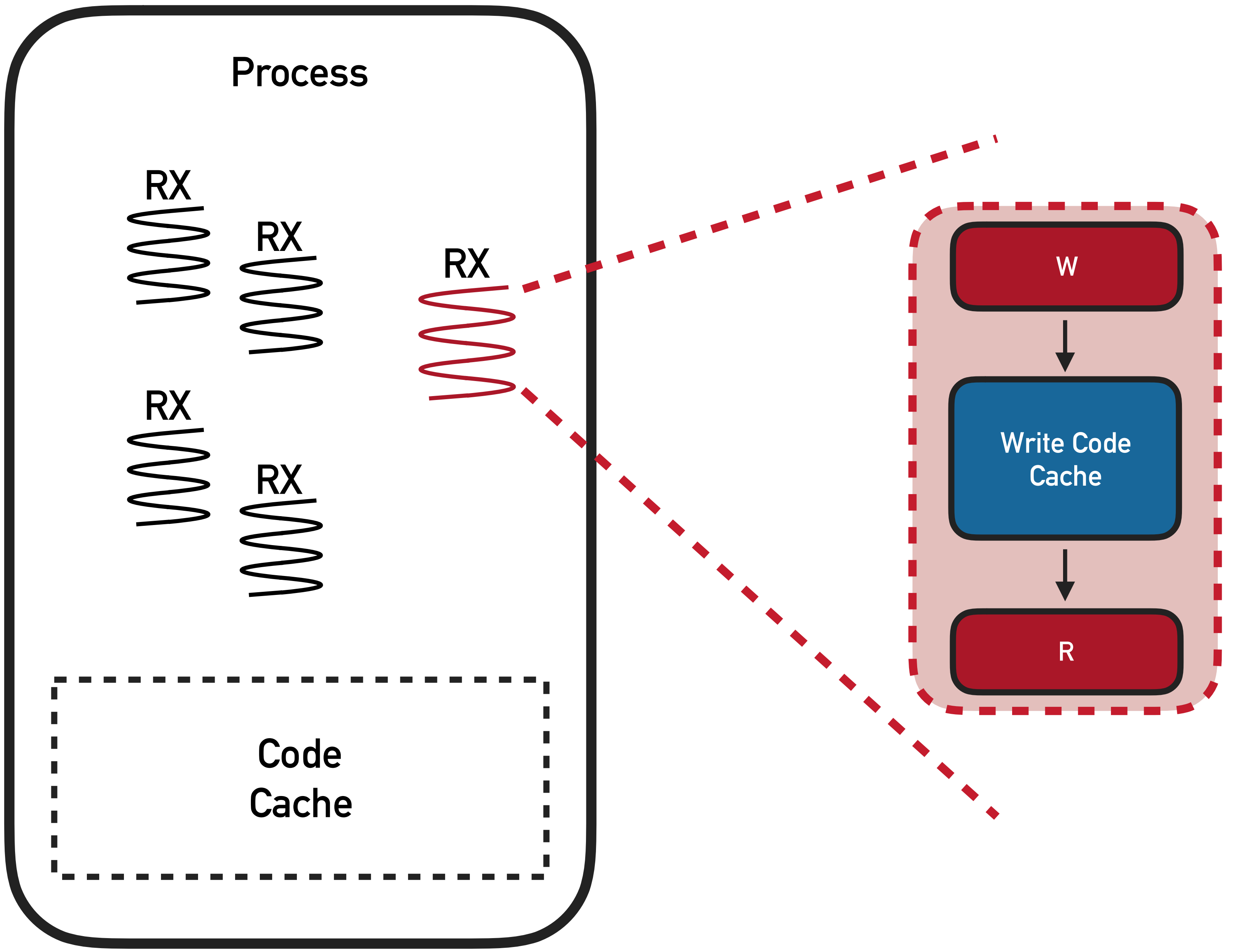
NON-SCALABLE HARDWARE RESOURCE

- Only 16 keys are provided



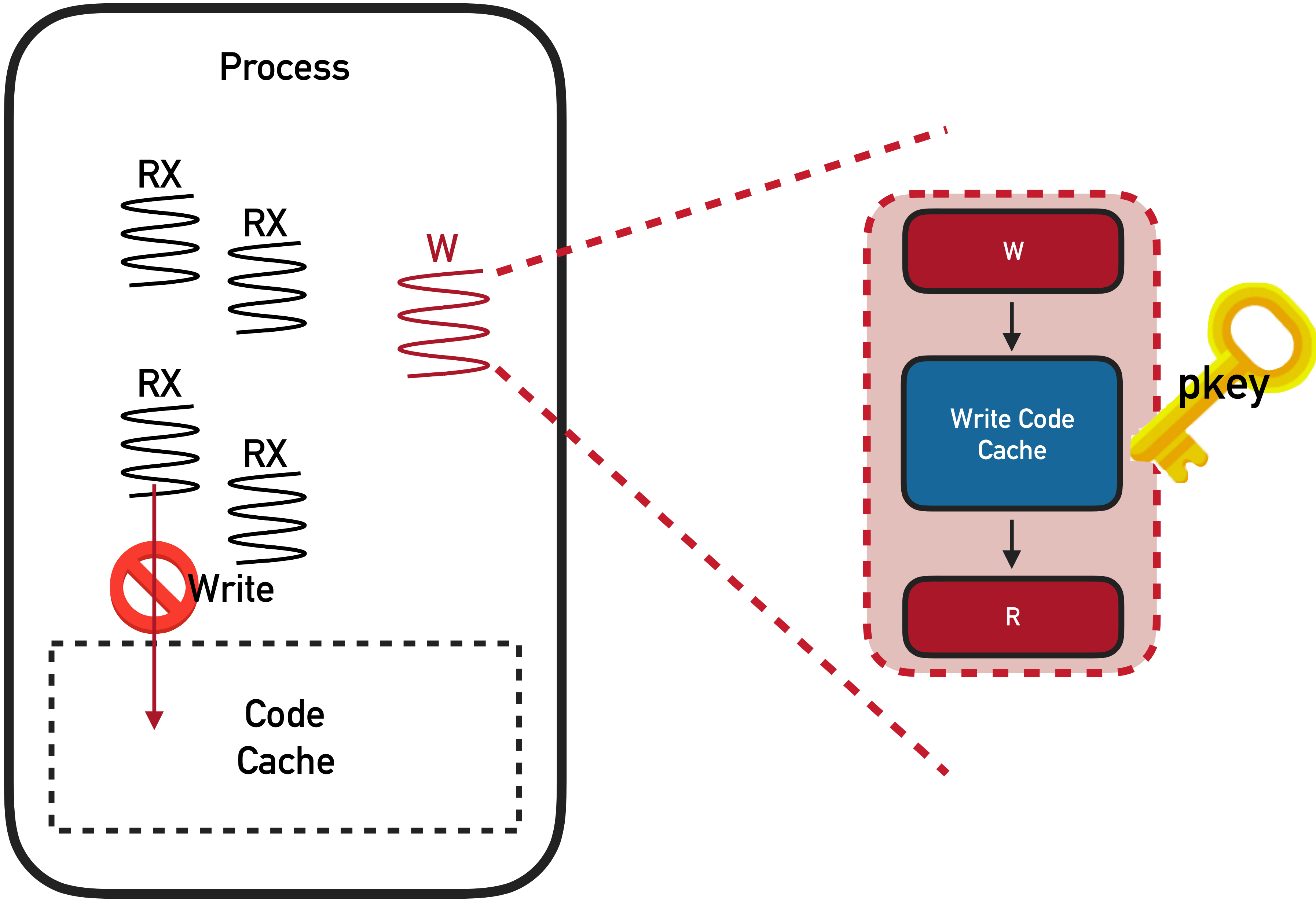
ASYNCHRONOUS PERMISSION CHANGE – PROS

- Permission change with MPK is per-thread intrinsically



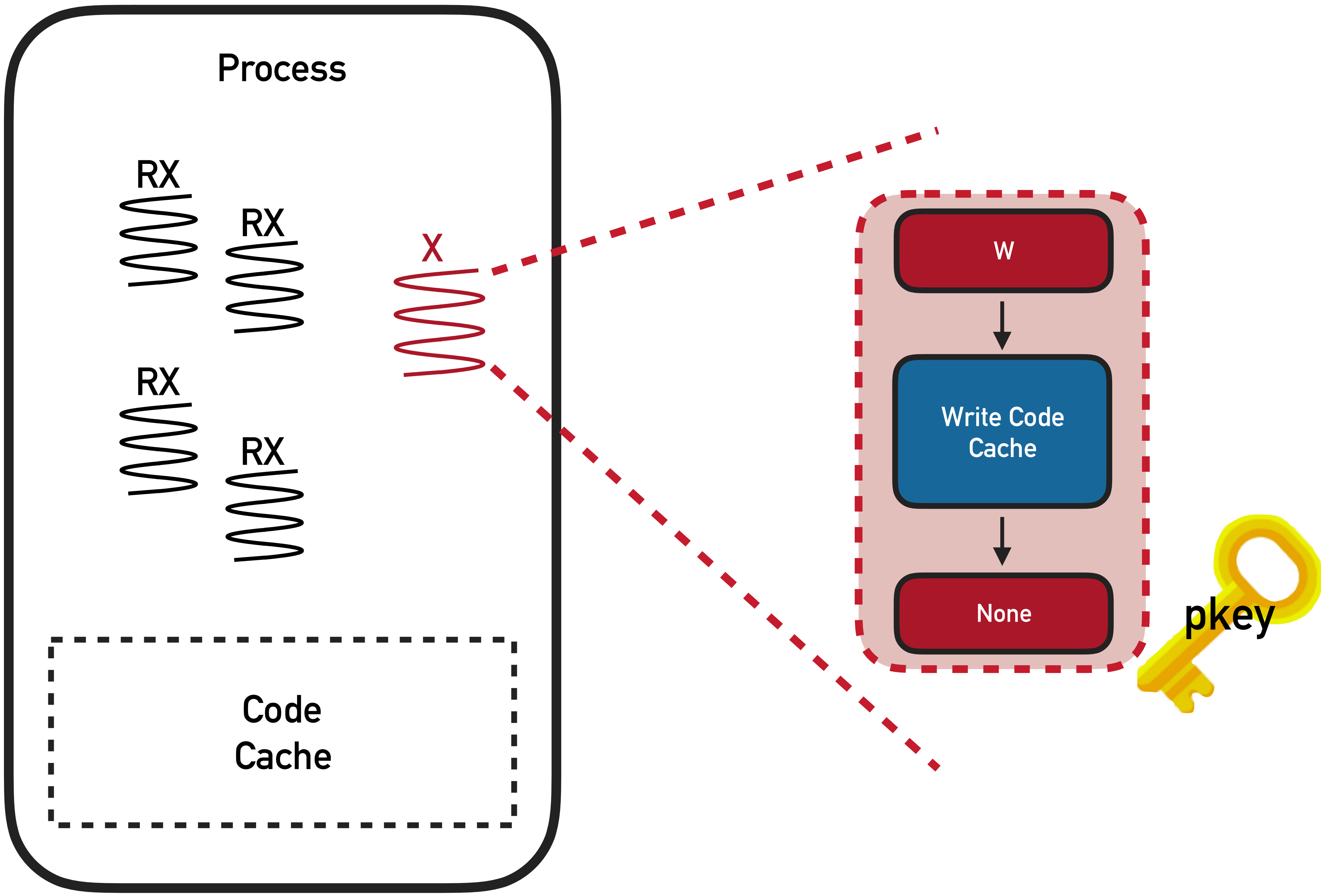
ASYNCHRONOUS PERMISSION CHANGE – PROS

- Permission change with MPK is per-thread intrinsically



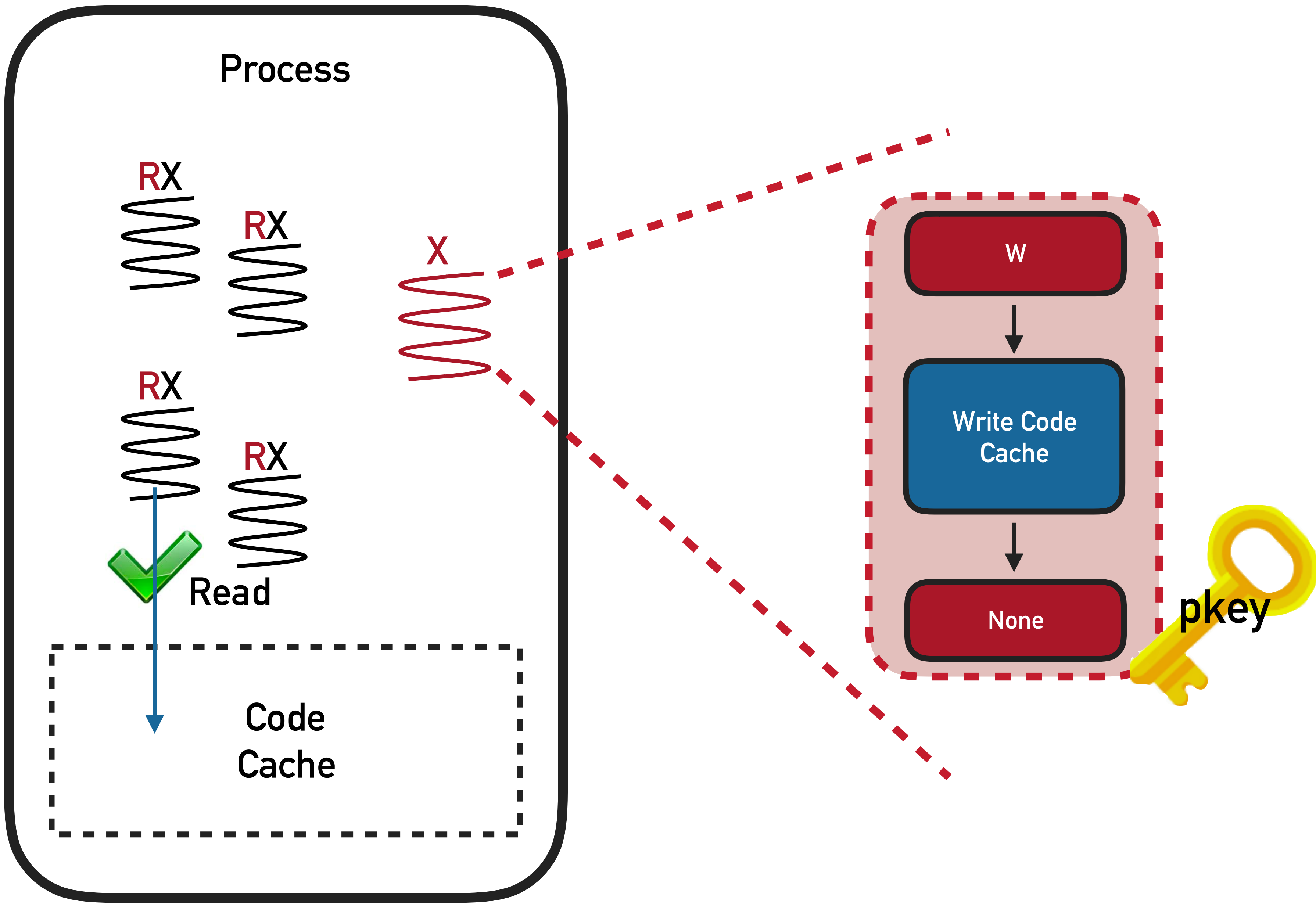
ASYNCHRONOUS PERMISSION CHANGE - CONS

- Permission synchronization is necessary in some context



ASYNCHRONOUS PERMISSION CHANGE – CONS

- Permission synchronization is necessary in some context



REVISIT : CHALLENGES

- Non-scalable Hardware Resources

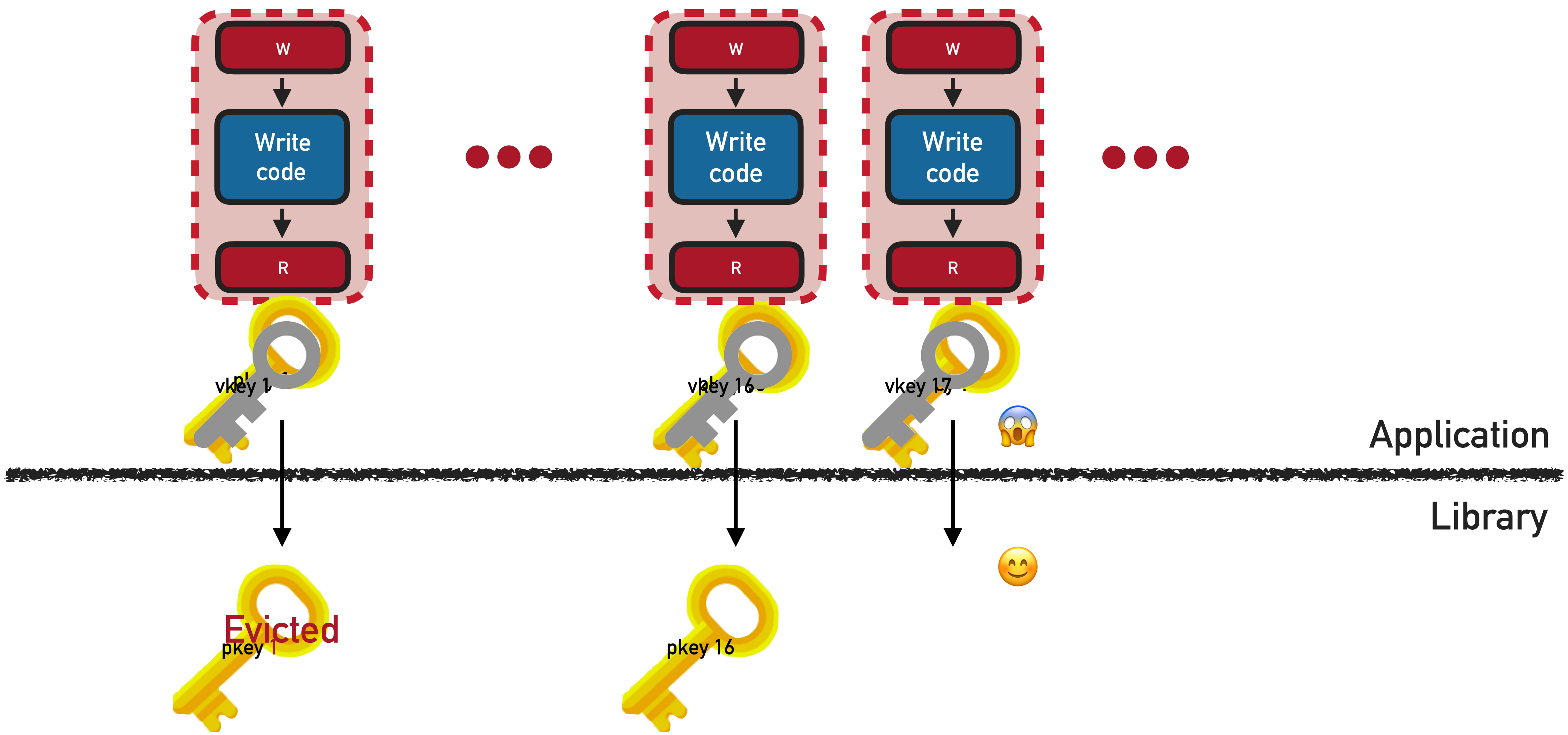
Key virtualization solve by key indirection.

- Asynchronous Permission Change

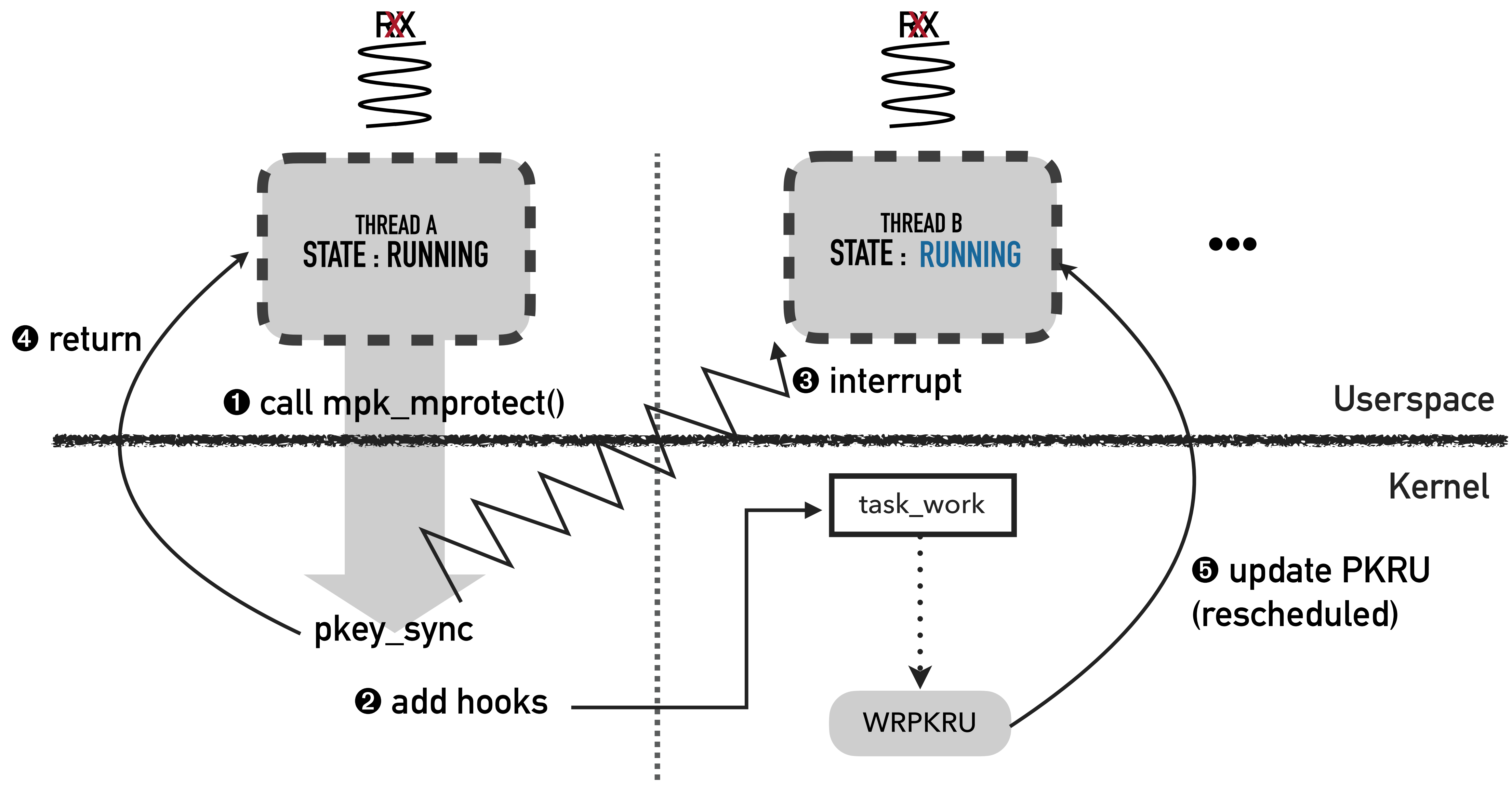
libmpk provide permission synchronization API

KEY VIRTUALIZATION

- Decoupling physical keys from user interface
  - Key indirection working like cache



INTER-THREAD PERMISSION SYNCHRONIZATION



IMPLEMENTATION

- libmpk is written in C/C++
  - Userspace library : 663 LoC
  - Kernel support : 1K LoC
    - Permission Synchronization
    - Kernel module for managing metadata
      - Userspace cannot fabricate metadata

- We open source at
  https://github.com/sslab-gatech/libmpk

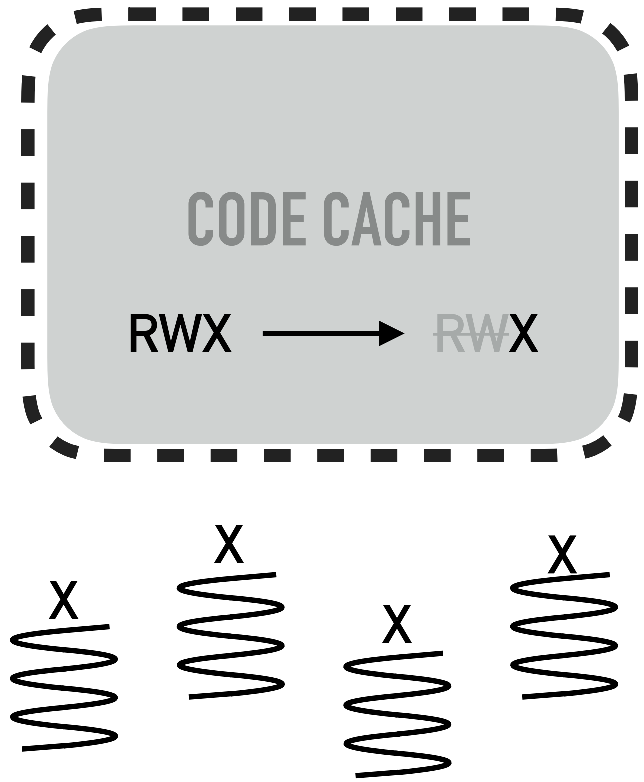
USE CASE – JIT PAGE W^X PROTECTION

```
function init()
  vkey = libmpk_mmap(&code_cache, len, RWX)
```

→ Key virtualization

```
function JIT()
  libmpk_begin(vkey, W)
  ...
  write code cache
  ...
  libmpk_end(vkey)
  libmpk_mprotect(vkey, X)
```

→ Permission synchronization



OUTLINE

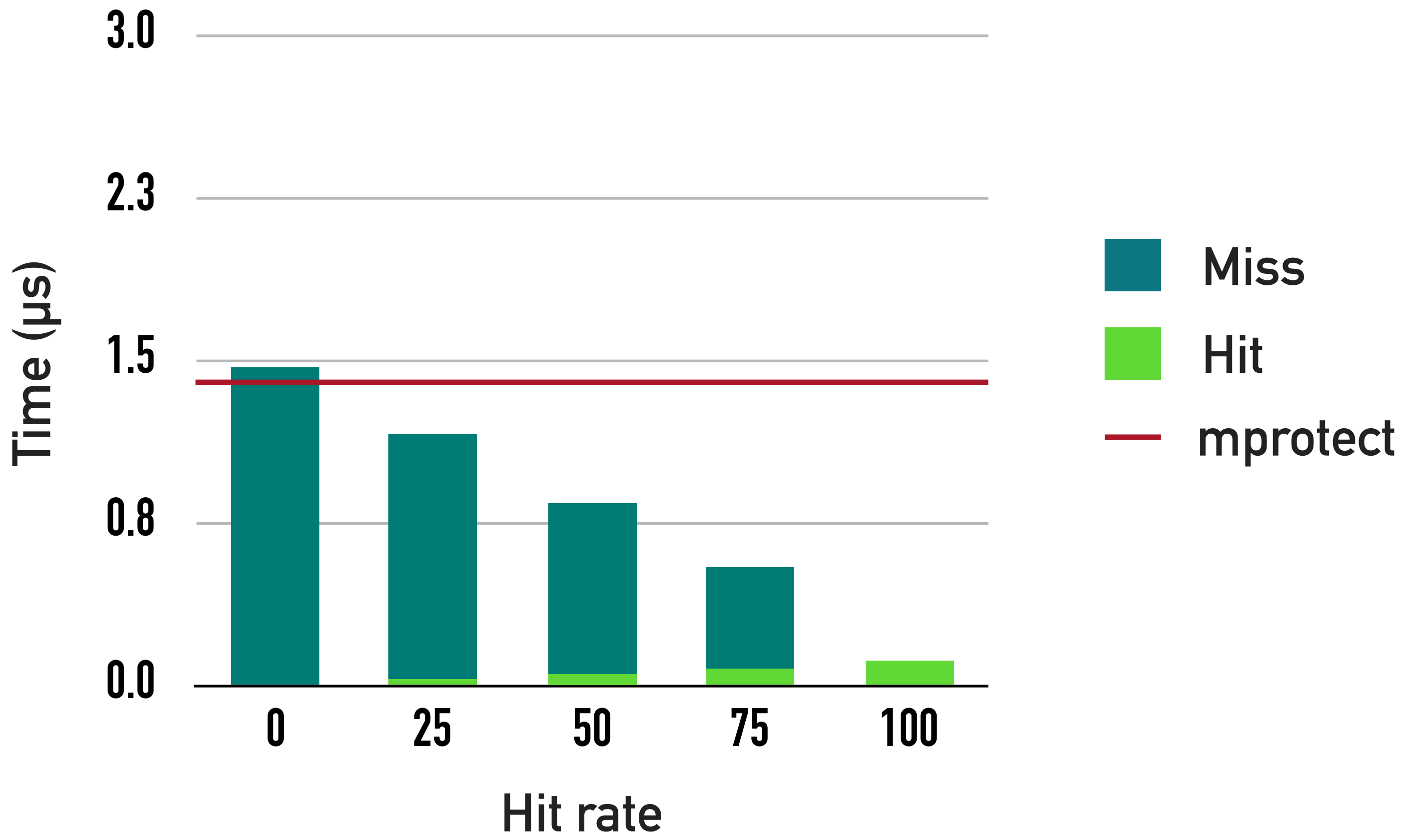
- Introduction
- Intel MPK Explained
- Challenges
- Design
- Implementation
- **Evaluation**
  - **Usability**
  - **Checking overhead occurred by design**
  - **Use cases - applying for memory isolation and protection**
- Discussion
- Related Work
- Conclusion

LIBMPK IS EASY TO ADOPT

- OpenSSL (83 LoC) : protecting private key
- Memcached (117 LoC) : protecting slabs
- Chakracore (10 LoC) : protecting JIT pages

LATENCY – KEY VIRTUALIZATION

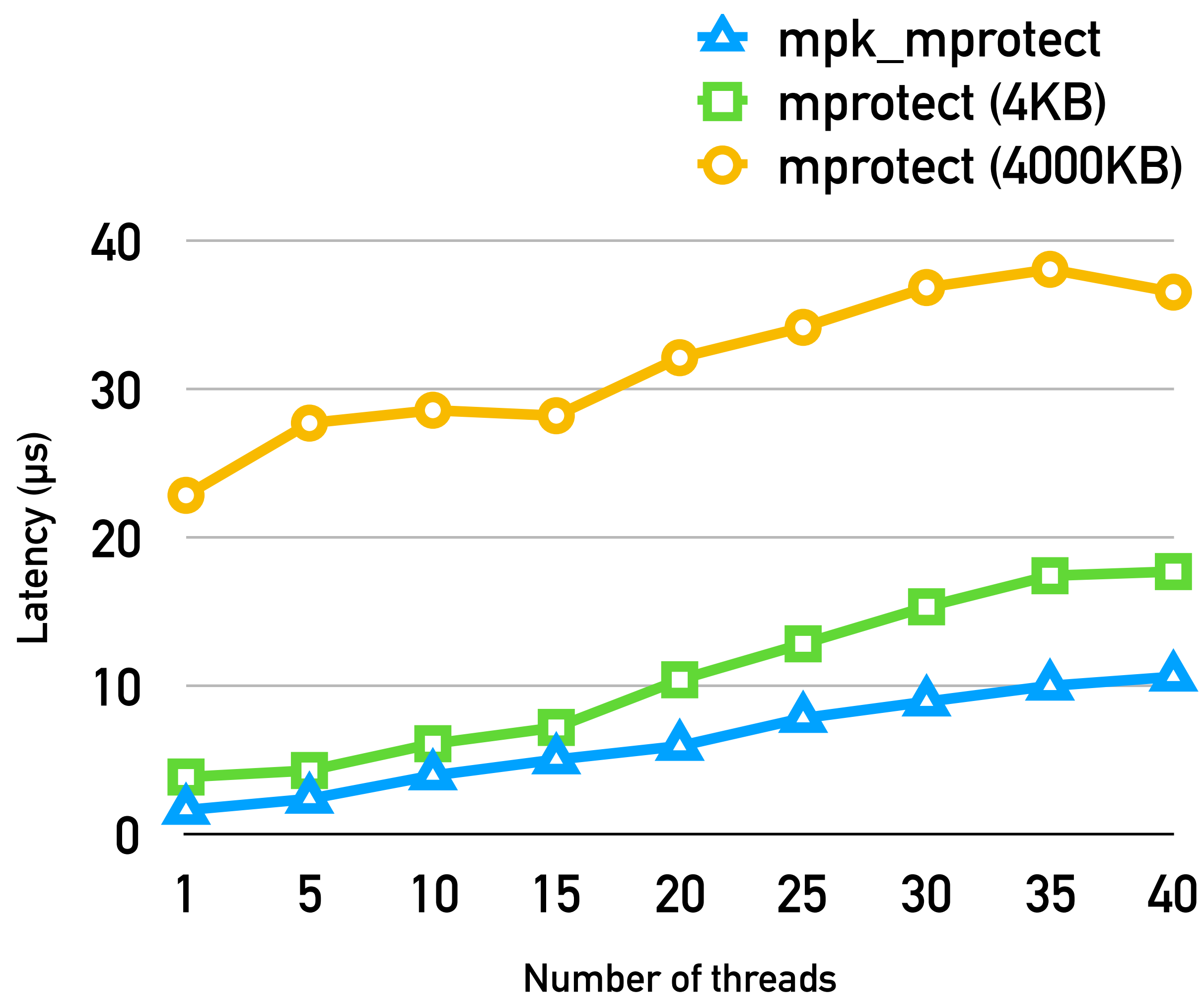
- Cache miss costs overhead due to eviction



Reasonable overhead while providing similar functionality.

LATENCY - INTER-THREAD PERMISSION SYNCHRONIZATION

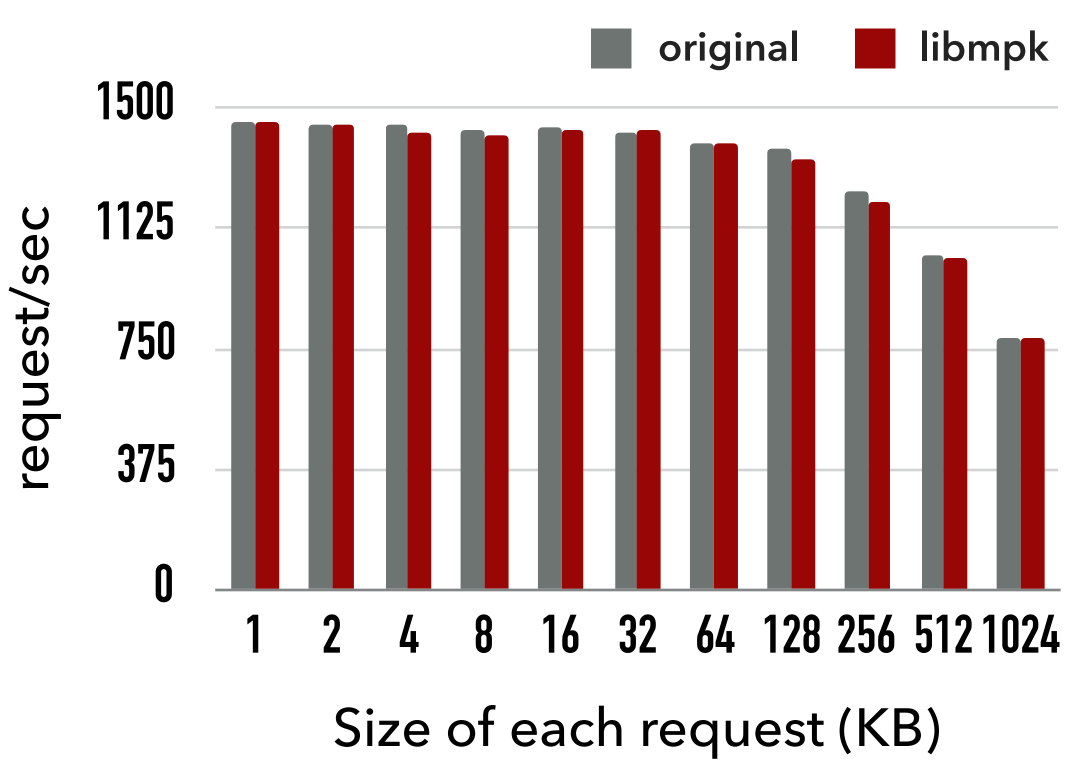
- Performance
  - 1,000 pages : 3.8x
  - Single page : 1.7x

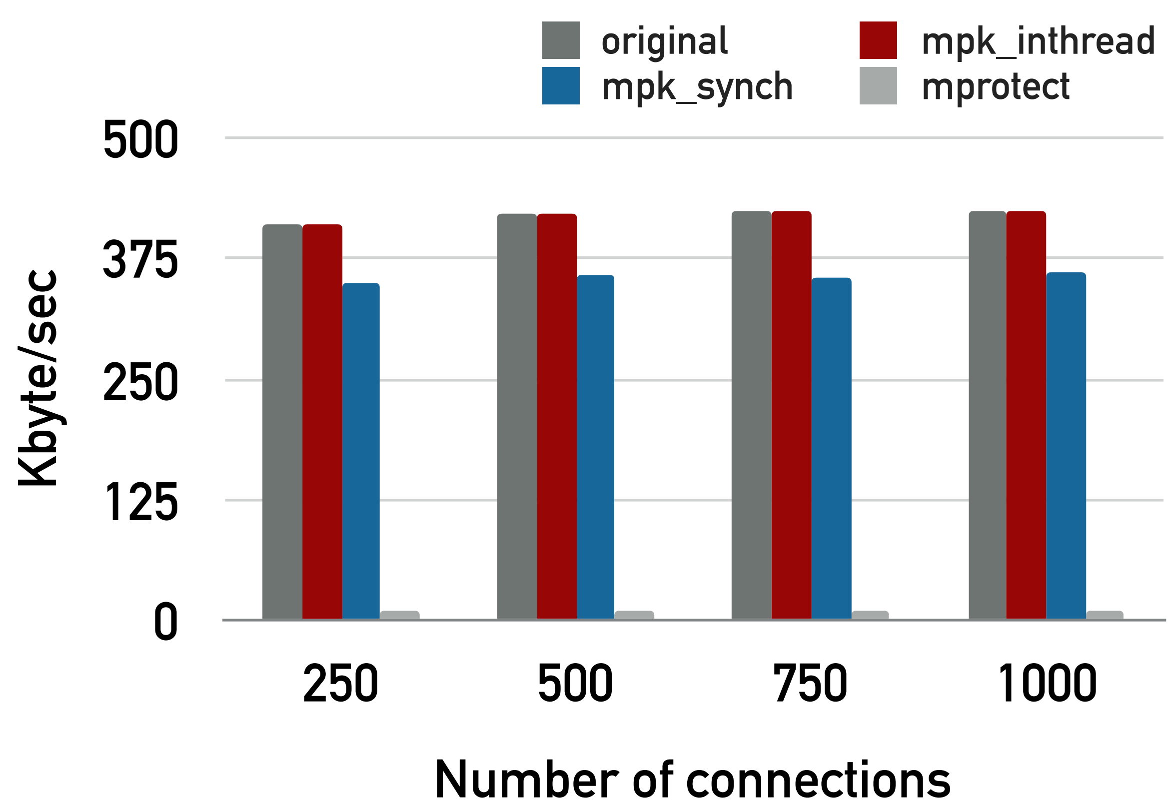


libmpk outperform mprotect regardless of the number of pages.

FAST MEMORY ISOLATION – OPENSSL & MEMCACHED

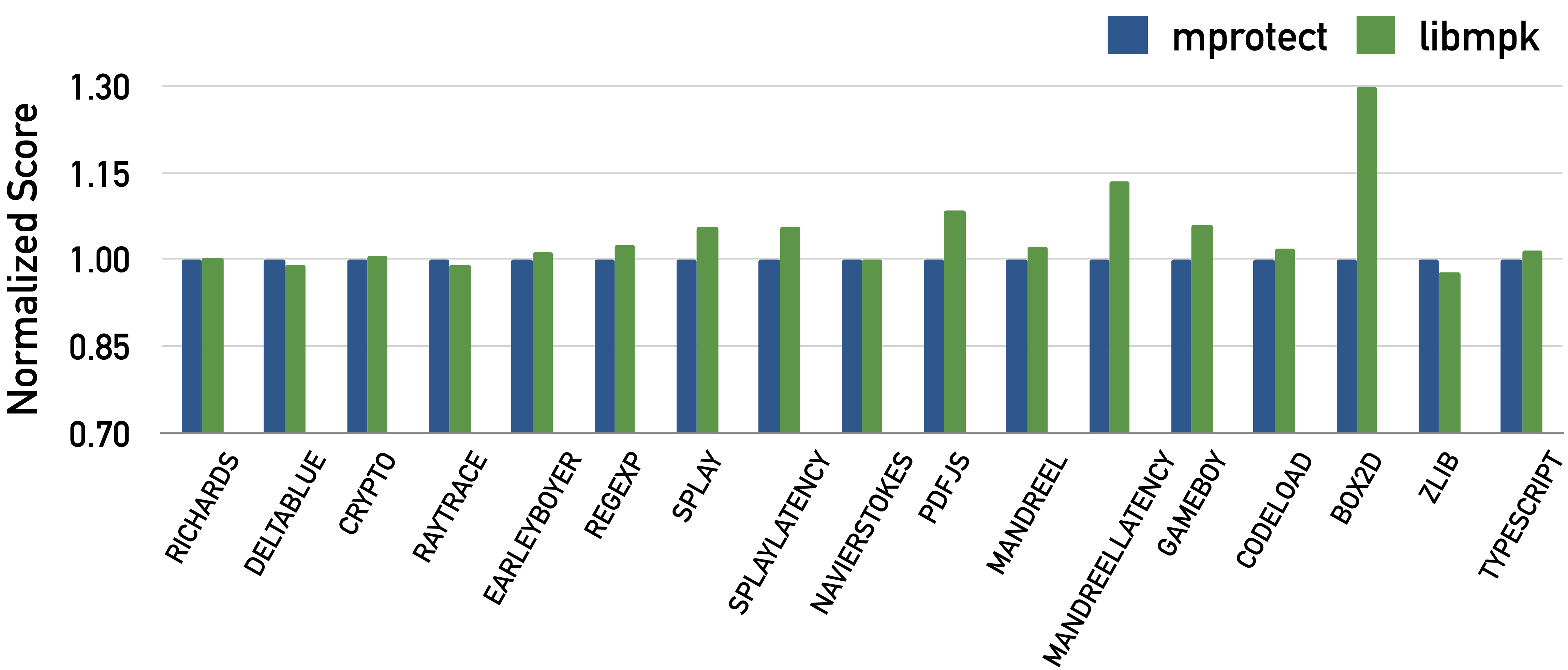
- OpenSSL
  - request/sec: 0.53% slowdown
- For 1GB protection :
  - original vs mpk_inthread : 0.01%
  - mpk_synch vs mprotect : 8.1x





FAST AND SECURE W⊕X - JIT COMPILATION

- Chakracore
  - mprotect-based protection
    - Allows race-condition attack
  - 4.39% performance improvement (31.11% at most)



DISCUSSION

- Rogue data cache load (Meltdown)
  - MPK is also affected by the Meltdown attack
  - Hardware or software-level mitigation
- Code reuse attack
  - Arbitrary executed WRPKRU may break the security
  - Applying sandboxing or control-flow integrity
- Protection key use-after-free
  - pkey_free does not perfectly free the protection key
  - Pages are still associated with the pkey after free

RELATED WORK

- ERIM [1] : Secure wrapper of MPK
- Shadow Stack [2] : Shadow stack protected by MPK
- XOM-Switch [3] : Code-reuse attack prevention with execute-only memory supported by MPK

[1] Anjo Vahldiek-Oberwagner, et al. "ERIM: Secure, Efficient In-Process Isolation with Memory Protection Keys", Security 2019
[2] Nathan Burow, et al. "Shining Light on Shadow Stacks", Oakland 2019
[3] Mingwei Zhang, et al. "XOM-Switch: Hiding Your Code From Advanced Code Reuse Attacks in One Shot", Black Hat Asia 2018

CONCLUSION

- *libmpk is a **secure, scalable, and synchronizable** abstraction of MPK for supporting fast memory protection and isolation with little effort.*

THANKS!

https://github.com/sslab-gatech/libmpk