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

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

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]

EXAMPLE 1 - HEARTBLEED ATTACK

Private key

Heartbleed request

Leaked data including private key

Web Server
EXAMPLE 1: EXISTING SOLUTION TO PROTECT MEMORY

- Process separation

EXAMPLE 2 - EXISTING SOLUTION TO PROTECT JIT PAGE

- JIT page $W^X$ protection
INTRODUCTION

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

![Graph showing latency (ms) vs. number of pages for mprotect (contiguous) and mprotect (sparse).]
INTEL MPK EXPLAINED

UNDERLINE IMPLEMENTATION

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

<table>
<thead>
<tr>
<th>page #</th>
<th>pkey</th>
<th>perm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>2</td>
<td>R/W</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

< Page table>

Kernel

pkey 2 <- R/W
page 120 -> R/W

pkey 2 <- R
page 120 -> R

32-bit register

16 pkeys

WRPKRU
RD PKRU
pkey_mprotect
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)
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

```python
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

Diagram:
- Process flow with read (RX) and write (W) operations
- Cache with read (RX) and write (W) permissions
- Key symbol (pkey)
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

![Diagram showing key virtualization process with write and read operations and key evictions.]
INTER-THREAD PERMISSION SYNCHRONIZATION

1. call mpk_mprotect()
2. add hooks
3. interrupt
4. return
5. update PKRU (rescheduled)
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](https://github.com/sslab-gatech/libmpk)
function init()
    vkey = libmpk_mmap(&code_cache, len, RWX)

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

Key virtualization

Permission synchronization

USE CASE - JIT PAGE $W^X$ PROTECTION
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

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