KShot: Live Kernel Patching with SMM and SGX

IEEE/IFIP DSN 2020 (Runner-up Best Paper Award)

Lei Zhou\textsuperscript{1,2}*, Fengwei Zhang\textsuperscript{1,2}**, Jinghui Liao\textsuperscript{3}, Zhenyu Ning\textsuperscript{1}, Jidong Xiao\textsuperscript{4}, Kevin Leach\textsuperscript{5}, Westley Weimer\textsuperscript{5}, Guojun Wang\textsuperscript{6}

\textsuperscript{1} SUSTech, China
\textsuperscript{2} Central South University, China
\textsuperscript{3} Wayne State University, USA
\textsuperscript{4} Boise State University, USA
\textsuperscript{5} University of Michigan, USA
\textsuperscript{6} Guangzhou University, China

* Work was done while visiting COMPASS lab at SUSTech; ** The corresponding author
Outline

• Introduction and Background
• Architecture of KShot
• Design and Implementation
• Evaluation: Effectiveness and Performance
• Conclusion
Why Need Patch the Kernel

Vulnerabilities cause attacks. Need to fix!
Patching Mechanism

Traditional Update
- WINDOWS UPDATE
- Ubuntu Updater
- RED HAT
- yum

Live Patching
- ORACLE Ksplice
- Canonical Livepatch
- kpatch
- kGraft
- debian
- SUSE

Users may unwilling to stop the runtime system, even need to patch kernel.

So, they choose kernel-based live patching.

But what if the kernel is compromised?
Challenges: Security

1. To patch the kernel, need to trust the kernel first!

   That’s a trap if the compromised kernel is against the patching!
Challenges: Resource Overhead

2. Overhead on Live patching may be larger than Restart

*Kernel-based Live Patching needs to store and restore the current system state*
Reliable Solution

Using Trusted and Isolated Execution Environment live patches the kernel without interrupting the target system!
TEE Background: SGX and SMM

Intel SMM system:
Strong isolation, which cannot be accessed by Host

Intel SGX system:
Provide Trust Environment in User level

Software

User mode
Supervisor mode

Hypervisor
 kernel

Hardware

DRAM
SMRAM
EPC

Main CPU
Intel Chipset
SGX CPU
Outline

- Introduction and Background
- Architecture of KShot
- Design and Implementation
- Evaluation: Effectiveness and Performance
- Conclusion
High-level Architecture of KShot
Outline

• Introduction and Background
• Architecture of KShot
• Design and Implementation
• Evaluation: Effectiveness and Performance
• Conclusion
SGX-based Patch Preparation

1 Reserve an isolated memory space.

2 Design a pre-preparing module In SGX enclave.
SGX-based Patch Preparation

4 Modify the effected instruction: like `branch`.

5 Final patch was encrypted and sent to reserved share memory.

3 Check the input binary patch.

Binary Code

```
<SYSC_kill>:
55
89 e5
...
E8 0C 74 35 00
...
: ftrace instruction
```

```
BF FD FF FF FF
74 3D
B8 00 00 00 00
E8 FC FF FF FF
83 FE FF
74 70
85 F6
...
C3
```

: begin of `sysc_kill`

: instructions from `kill_something_info`

: end of `sysc_kill`
SMM-based Live Patching

The workflow of patching in SMM handler.

Also, it is easy to rollback and update the patch with the similar operations.
Outline

• Introduction and Background
• Architecture of KShot
• Design and Implementation
• Evaluation: Effectiveness and Performance
• Conclusion
Evaluation

The test environment platform:

- real-world patches from the Common Vulnerabilities and Exposures (CVE) Database.
- analyzed 267 such vulnerabilities for Linux kernels 3.14 and 4.4.
- Intel Core i7 CPU (supporting SGX and SMM) with 16GB memory.
- a combination of Coreboot with a SeaBIOS payload as the system BIOS.
While deploying KShot, we consider three research questions:

- **RQ1.** Can KShot correctly apply kernel patches?
- **RQ2.** What is KShot’s performance overhead?
- **RQ3.** How does KShot compare to existing approaches?
RQ1. Can KShot correctly apply kernel patches?

We randomly selected 30 of those 214 patches.
Part of experimental results shown in above table.

KShot can correctly apply kernel patches.
RQ2. What is KShot’s performance overhead?

- SGX-based pre-preparation introduces extra overhead, but does not interrupt the normal system.
- SMM-based patching causes a very short pause, and the normal system state stays the same.

**Time overhead in each step of real CVE case live patching**

- SGX-based patch preparation time.
- SMM-based live patching time.
RQ3. How does KShot compare to existing approaches?

<table>
<thead>
<tr>
<th></th>
<th>Kernel Dependency</th>
<th>Untrusted OS</th>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dyninst [24]</td>
<td>✓</td>
<td>✗</td>
<td>userspace</td>
</tr>
<tr>
<td>EEL [10]</td>
<td>✓</td>
<td>✗</td>
<td>userspace</td>
</tr>
<tr>
<td>Libcare [25]</td>
<td>✓</td>
<td>✗</td>
<td>userspace</td>
</tr>
<tr>
<td>Kitsune [59]</td>
<td>✓</td>
<td>✗</td>
<td>userspace</td>
</tr>
<tr>
<td>PROTEOS [26]</td>
<td>✓</td>
<td>✗</td>
<td>kernel</td>
</tr>
<tr>
<td>KSHOT</td>
<td>✗</td>
<td>✓</td>
<td>kernel</td>
</tr>
</tbody>
</table>

We can find that only KShot is kernel independent and useable in Untrusted OS.
RQ3. How does KShot compare to existing approaches?

Comparison with kernel patching systems.

<table>
<thead>
<tr>
<th>Type</th>
<th>Downtime</th>
<th>Untrusted OS</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>KUP [8]</td>
<td>kernel 3s/kernel</td>
<td>✗</td>
<td>&gt;30G</td>
</tr>
<tr>
<td>KARMA [9]</td>
<td>instruction 5μs/patch¹</td>
<td>✗</td>
<td>lua engine</td>
</tr>
<tr>
<td>kpatch [10]</td>
<td>function 45.6ms/patch¹</td>
<td>✗</td>
<td>16G</td>
</tr>
<tr>
<td><strong>KSHOT</strong></td>
<td>function 50μs/patch¹</td>
<td>✓</td>
<td>18M</td>
</tr>
</tbody>
</table>

¹ for an average-sized patch of less than 1KB

The performance of KShot is better
Outline

• Introduction and Background
• Architecture of KShot
• Design and Implementation
• Evaluation: Effectiveness and Performance
• Conclusion
Conclusion

＊ **KShot - secure and efficient framework for kernel patching**
  — Leveraging Intel SMM.
  — Leveraging Intel SGX.
  — Against indicative kernel vulnerabilities.

＊ **Application scenarios**
  — Compromised Hypervisor, OS kernels.
  — Without external checkpoint-and-restore resources.

＊ **Introducing low overhead and a small trusted code base**
Thank You for Your Attention! 
Questions?

zhangfw@sustech.edu.cn

http://cse.sustech.edu.cn/faculty/~zhangfw/
Backup Slides
KShot Design & Implementation

- **Binary Patch Preparation**
- SGX-based Patch Preparation
- SMM-based Live Patching
- Patching Protection
Identify the Patch Function

We assume we can get the trusted patch source code.

Vulnerable functions are defined with three types:
- Type 1: non-inline function,
- Type 2: inline function,
- Type 3: special case: data structure changed function.

Finding the final target function for patching is different in each type.
Target Function Analysis

With knowing a vulnerable function, need to find the patching function:

1. Get the binary kernel code through compiling the kernel source.
2. Locate the vulnerable instruction segments.
3. Identify the patching-needed function.
KShot Design & Implementation

- Binary Patch Preparation
- SGX-based Patch Preparation
- SMM-based Live Patching
- Patching Protection
Patching Protection

Malicious Patch Reversion

• SMM-based kernel protection.
• Introspect regions of memory overwritten with trampoline instructions.

Denial-of-service attacks

• Generally difficult to defend.
• Identify the memory written events with SMM and remote server.