SCONE: Secure Linux Container Environments with Intel SGX

S. Arnautov, B. Trach, F. Gregor, Thomas Knauth, and A. Martin, Technische Universität Dresden; C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, and M. Stillwell, Imperial College London; D. Goltzsche, Technische Universität Braunschweig; D. Eyers, University of Otago; R. Kapitza, Technische Universität Braunschweig; P. Pietzuch, Imperial College London; C. Fetzer, Technische Universität Dresden

thomas.knauth@tu-dresden.de
Trust Issues: The Provider’s Perspective

- Cloud provider does not trust users
- Use virtual machines to isolate users from each other and the host
- VMs only provide one way protection
Trust Issues: The User’s Perspective

• Users trust their application

• Users must implicitly trust the cloud provider

• Existing applications implicitly assume trusted operating system
Containers are the new VMs

- Containers provide resource isolation and bundling
- Smaller resource overhead than virtual machines
- Convenient tooling to create and deploy applications in the cloud
Disaster!
Disaster!
Disaster!

- OS
- VMM
- Firmware
- Cloud platform
- Staff
  ...
  untrusted
Disaster!
We want to ...
We want to ...

- run unmodified Linux applications ...
We want to …

- run unmodified Linux applications …
- in containers …
We want to …

- run unmodified Linux applications …
- in containers …
- in an untrusted cloud …
We want to …

- run unmodified Linux applications …
- in containers …
- in an untrusted cloud …
- securely and …
We want to …

- run unmodified Linux applications …
- in containers …
- in an untrusted cloud …
- securely and …
- with acceptable performance
Secure Guard Extensions

- New **enclave** processor **mode**
- Users can create a HW-enforced trusted environment
- Only trust Intel and Secure Guard Extensions (SGX) implementation
SGX: HW-enforced Security

- 18 new instructions to manage enclave life cycle
- **Enclave memory** only accessible from enclave
- Certain instructions disallowed, e.g., `syscall`
Challenge 1: Interface

• Haven (OSDI’14): library operating system in enclave

• Large TCB → more vulnerable

• Small interface (22 system calls)

• Shields protect the interface
Challenge 1: Interface

- Small TCB
- C library interface is complex
- Harder to protect

Diagram:
- Minimal TCB
  - Application Code
  - Libraries
  - Shim C Library
- C Library
- Host OS
Challenge 2: Performance

- pwrite() with 32 byte buffer
- 4 cores with hyper threading

Graph showing system call frequency (1000s/second) vs. Threads:
- Native
- Synchronous enclave exits

The graph illustrates the performance impact of increasing the number of threads.
Challenge 2: Performance

- pwrite() with 32 byte buffer
- 4 cores with hyper threading
SCONE Architecture

Application
Libraries

SCONE module
Intel SGX driver
Container (cgroups)
Host operating system
SCONE Architecture

- Enhanced C library → small TCB (Challenge 1)
SCONE Architecture

- Enhanced C library → small TCB (Challenge 1)
- Asynchronous system calls and user space threading reduce number of enclave exits (Challenge 2)
SCONE Architecture

- Enhanced C library → small TCB (Challenge 1)
- Asynchronous system calls and user space threading reduce number of enclave exits (Challenge 2)
- Network and file system shields actively protect user data

Diagram:

- Application
- Libraries
  - Network shield
  - File system shield
  - M:N threading
  - SCONE C library
  - Asynchronous system calls

- SCONE module
- Intel SGX driver
- Container (cgroups)
- Host operating system
Anatomy of a System Call
Anatomy of a System Call

\{ T1 \}

\textbf{read}(\text{fd}, \text{buf}, \text{size})

\begin{itemize}
  \item enclave
  \item kernel
\end{itemize}

\begin{itemize}
  \item [0]
  \item [1]
  \item [2]
\end{itemize}

\textbf{system call slots}
Anatomy of a System Call

\[ \text{read(fd, buf, size)} \]
Anatomy of a System Call

\{ T1 \}
read(fd, buf, size)

[0]
[1]
[2]

system call slots

enclave

kernel
Anatomy of a System Call

```
T1
```

read(fd, buf, size)
Anatomy of a System Call

\[
\{ T1 \} \quad \text{read}(fd, \text{buf}, \text{size})
\]

enclave

kernel

system call slots

\[
\begin{array}{c}
[0] \quad \text{read, fd, buf, size} \\
[1] \\
[2]
\end{array}
\]
Anatomy of a System Call

read(fd, buf, size)

switch to ready
user space thread

[0]
read, fd, buf, size

[1]

[2]
system call slots

[enclave]

kernel
Anatomy of a System Call

switch to ready user space thread

enclave

kernel

system call slots
Anatomy of a System Call

read(fd, buf, size)

switch to ready user space thread

[0] read, fd, buf, size
[1]
[2] read, fd, buf, size

system call slots
Anatomy of a System Call

```
T1
read(fd, buf, size)

T2
read(fd, buf, size)
```

```
[0] read, fd, buf, size
[1]  
[2] read, fd, buf, size
```

enclave

kernel

system call slots
Anatomy of a System Call

T1
read(fd, buf, size)

T2
read(fd, buf, size)

enclave
kernel

#2&$??%

system call slots
Anatomy of a System Call

T1

read(fd, buf, size)

T2

read(fd, buf, size)

switch to ready user space thread

[0] read, fd, buf, size
[1] 
[2] read, fd, buf, size

system call slots

enclave

kernel

#2&$??%
Anatomy of a System Call

```
T1
read(fd, buf, size)
```

```
T2
read(fd, buf, size)
```

```
[0]
[1]
[2]
```

- System call slots
- Enclave
- Kernel

```
[0] read, fd, buf, size
[1] 
[2] read, fd, buf, size
```

```
#2&$??%
```
Anatomy of a System Call

T1
GET K1
read(fd, buf, size)

decrypt buffer into enclave

T2
read(fd, buf, size)

[0] read, fd, buf, size
[1]  
[2] read, fd, buf, size

system call slots

enclave

kernel

#2&$??%
Container Integration

- Repository
- Docker Engine
- Secure Image
- SCONE Client
- Docker Client
- Enclave
Container Integration

1. push image

Secure Image

SCONE Client
Docker Client

Repository

Docker Engine

Enclave
Container Integration

1. push image

- Secure Image
- SCONE Client
- Docker Client

Repository

2. run

Docker Engine

Enclave
Container Integration

1. push image
2. run
3. pull image

Repository

Docker Engine

Secure Image

SCONE Client

Docker Client

Enclave
Container Integration

1. push image
2. run
3. pull image
4. execute

Repository

Docker Engine

Secure Image

SCONE Client

Docker Client

Enclave
Container Integration

1. push image
2. run
3. pull image
4. execute
5. secure channel

Repository

Docker Engine

Secure Image

SCONE Client

Docker Client

Enclave
System Call Performance

- pwrite() with 32 byte buffer
- 4 cores with hyper threading
System Call Performance

- async with 1 thread achieves 80%
- pwrite() with 32 byte buffer
- 4 cores with hyper threading
System Call Performance

Async with 1 thread achieves 80%

Optimized queue may help

- `pwrite()` with 32 byte buffer
- 4 cores with hyper threading
Apache Throughput

Latency (seconds)

Throughput (requests / second)

sync
async

glibc
## Performance Overview

<table>
<thead>
<tr>
<th>Application</th>
<th>Throughput w.r.t. native async (%)</th>
<th>sync (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached</td>
<td>120</td>
<td>113</td>
</tr>
<tr>
<td>Apache</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>NGINX</td>
<td>80</td>
<td>36</td>
</tr>
<tr>
<td>Redis</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>
## Performance Overview

<table>
<thead>
<tr>
<th>Application</th>
<th>Throughput w.r.t. native async (%)</th>
<th>Throughput w.r.t. native sync (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached</td>
<td>120</td>
<td>113</td>
</tr>
<tr>
<td>Apache</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>NGINX</td>
<td>80</td>
<td>36</td>
</tr>
<tr>
<td>Redis</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>

inline encryption has less overhead
## Performance Overview

<table>
<thead>
<tr>
<th>Application</th>
<th>Throughput w.r.t. native async (%)</th>
<th>sync (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached</td>
<td>120</td>
<td>113</td>
</tr>
<tr>
<td>Apache</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>NGINX</td>
<td>80</td>
<td>36</td>
</tr>
<tr>
<td>Redis</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>

- Inline encryption has less overhead.
- Inline encryption hurts performance with single thread.
Summary

- Small trusted computing base (0.6× – 2.0× of native binary size)
- Low runtime overhead (0.6× – 1.2× of native throughput)
- Transparent to the container engine (e.g. Docker)