Tarnhelm: Isolated, transparent and confidential execution of arbitrary code in ARM's TrustZone

Davide Quarta, Michele Ianni, Aravind Machiry, Yanick Fratantonio, Eric Gustafson, Davide Balzarotti, Martina Lindorfer, Giovanni Vigna, Christopher Kruegel
Overview

1. Introduction
2. Trusted Execution Environments
3. Design Goals
4. Approach
5. Implementation
7. Performance Evaluation
8. Conclusion
Introduction
Introduction

Applications running on a commodity operating system are usually deployed in an untrusted environment.

The user has full access to any of the application’s assets, *including its code*. 
Introduction

In the absence of architectural support to protect an application’s code from unauthorized access, thus avoiding intellectual property loss and piracy of paid content, developers have to rely on:

- Code obfuscation
- Anti-tampering and Anti-debugging techniques
- Different distribution strategies (e.g., in-app purchases)
“All intellectual property protection technologies will be cracked at some point - it’s just a matter of time”

- Microsoft
Can we achieve Code Confidentiality using *Trusted Execution Environments*?
Introduction

- TEEs operate on a higher level of privilege, they are only designed to execute trusted code signed by device vendors
- TEEs are resource-constrained and not designed to execute full-fledged applications

We address these challenges in Tarnhelm, which *transparently executes* individual code components in TrustZone and guarantees *code confidentiality* through isolation, without sacrificing overall system security.
Trusted Execution Environments
Trusted Execution Environment (TEE)

- Hardware-isolated execution environment (e.g., ARM TrustZone)
  - Non-secure world
    - Untrusted OS and untrusted applications (UAs) (e.g., Android and apps)
  - Secure world
    - Higher privilege, can access *everything*
    - Trusted OS and trusted applications (TAs)
ARM TrustZone

- CPU
- DMA
- AMBA 5 AHB5 Interconnect
- Flash
- SRAM
- Non-Trusted Peripheral A
- Trusted Peripheral B

[Diagram showing secure regions with locked icons indicating trusted and non-trusted areas.]
Limitations of Existing TEEs

Developers must

- manually partition an application’s code into a secure and non-secure part;
- define interfaces between the two parts;
- modify the secure code part to be compatible with the TEE.
Design Goals
Design Goals

- Code confidentiality
- Transparent forwarding
- Transparent integration
- Limited attack surface
- Minimal overhead
Approach
Deployment
Code Partitioning

```c
#include<stdio.h>
int curr_idx = 0;

#define __tarnhelm__attribute__((section(".invisible")))
__tarnhelm__ void* get_processed_data(struct object *data){
    void* get_processed_data(struct object *data){
        increment_counter(data);
        // use data to perform some computation
        return data;
    }
    void increment_counter(struct object *data){
        if(data != NULL){
            data->counter += curr_idx;
            curr_idx++;
        }
    }
}

int main(){
    struct object curr_data;
    ...
    get_processed_data(curr_data);
    ...
}
```
Secure Code Retrieval and Loading
# Memory Management

<table>
<thead>
<tr>
<th></th>
<th>Normal World</th>
<th>Secure World</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>0x0400</td>
<td>0x0500</td>
</tr>
<tr>
<td></td>
<td>0x1000</td>
<td>0x1100</td>
</tr>
<tr>
<td>.data</td>
<td>0x0500</td>
<td>0x0600</td>
</tr>
<tr>
<td></td>
<td>0x1100</td>
<td>0xE100</td>
</tr>
<tr>
<td>.invisible</td>
<td>0x0600</td>
<td>0x0600</td>
</tr>
<tr>
<td></td>
<td>0xE100</td>
<td>0xE100</td>
</tr>
<tr>
<td>stack</td>
<td>0x7F00</td>
<td>0x7F00</td>
</tr>
<tr>
<td></td>
<td>0x3000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>VA</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>0x0400</td>
<td>0x1000</td>
</tr>
<tr>
<td></td>
<td>0x1100</td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td>0x0500</td>
<td>0x1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.invisible</td>
<td>0x0600</td>
<td>0xE100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>0x7F00</td>
<td>0x3000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System Call Forwarding
Transparent World Switch
Implementation
Implementation

We implemented Tarnhelm based on the default OP-TEE 2.3.0 32-bit QEMU configuration. We added:

- 3.11K lines of code (LOC) to the TCB
- 1,415 LOC to the OP-TEE OS
- 566 LOC to the Linux abort handler and include files
- 1,129 LOC to the OP-TEE Linux driver
Transparent Execution
## Control-Flow Integrity

<table>
<thead>
<tr>
<th>↓From/To→</th>
<th>Untrusted OS</th>
<th>Trusted OS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Untrusted OS</strong></td>
<td>ret</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>call</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Trusted OS</strong></td>
<td>ret</td>
<td>Pop shadow stack</td>
</tr>
<tr>
<td></td>
<td>call</td>
<td>Push return address on the shadow stack</td>
</tr>
</tbody>
</table>
Security Evaluation
Attacks on Code Confidentiality

- Instruction inference attacks
- Control-flow redirection attacks
- Data-only attacks
- Iago attacks
- Blind ROP
- Vulnerabilities in the invisible code
- Compromised TA
- Emulated TEE
Performance Evaluation
Performance Evaluation

We evaluated Tarnhelm on QEMU emulating an ARMv7 Cortex-A15 with soft-mmu, running on an Intel Core 8-core i7-930 CPU (2.80GHz) desktop machine with 12GB of memory.
## Microbenchmark of Tarnhelm’s Individual Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invisible code initialization</td>
<td>0.316s</td>
</tr>
<tr>
<td>Invisible code cleanup</td>
<td>0.44ms</td>
</tr>
<tr>
<td>System call forwarding</td>
<td>116.88µs</td>
</tr>
<tr>
<td>Data mapping (secure world)</td>
<td>71µs</td>
</tr>
<tr>
<td>Data mapping (normal world)</td>
<td>231.337µs</td>
</tr>
<tr>
<td>IW-CFI indirect call (trusted OS)</td>
<td>0.111µs</td>
</tr>
<tr>
<td>IW-CFI return (trusted OS)</td>
<td>19.431µs</td>
</tr>
</tbody>
</table>
## Overhead of the Transparent World Switch

<table>
<thead>
<tr>
<th>Direction</th>
<th>w/ DM+IWCFI</th>
<th>w/ DM fwd</th>
<th>w/o DM fwd</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW → NW call → ret SKW</td>
<td>495.529μs</td>
<td>494.539μs</td>
<td>152.093μs</td>
</tr>
<tr>
<td>NW → SW call → ret SW</td>
<td>505.348μs</td>
<td>497.549μs</td>
<td>151.298μs</td>
</tr>
<tr>
<td>SW id-call → NW ret NW</td>
<td>514.903μs</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
LMBench Results

![Bar Chart showing LMBench Results](chart.png)
Macro Experiment with a Real-World Game
Conclusion
Conclusion

- Tarnhelm, an approach that offers a new powerful primitive: code confidentiality
- Transparent execution of parts of an unmodified application in different isolated execution environments
- Limited additions to the TCB
- Resiliency of Tarnhelm against potential attacks
- Reasonable performance overhead
- Open source, available at https://github.com/ucsb-seclab/invisible-code
Questions?