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
6. Security Evaluation
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

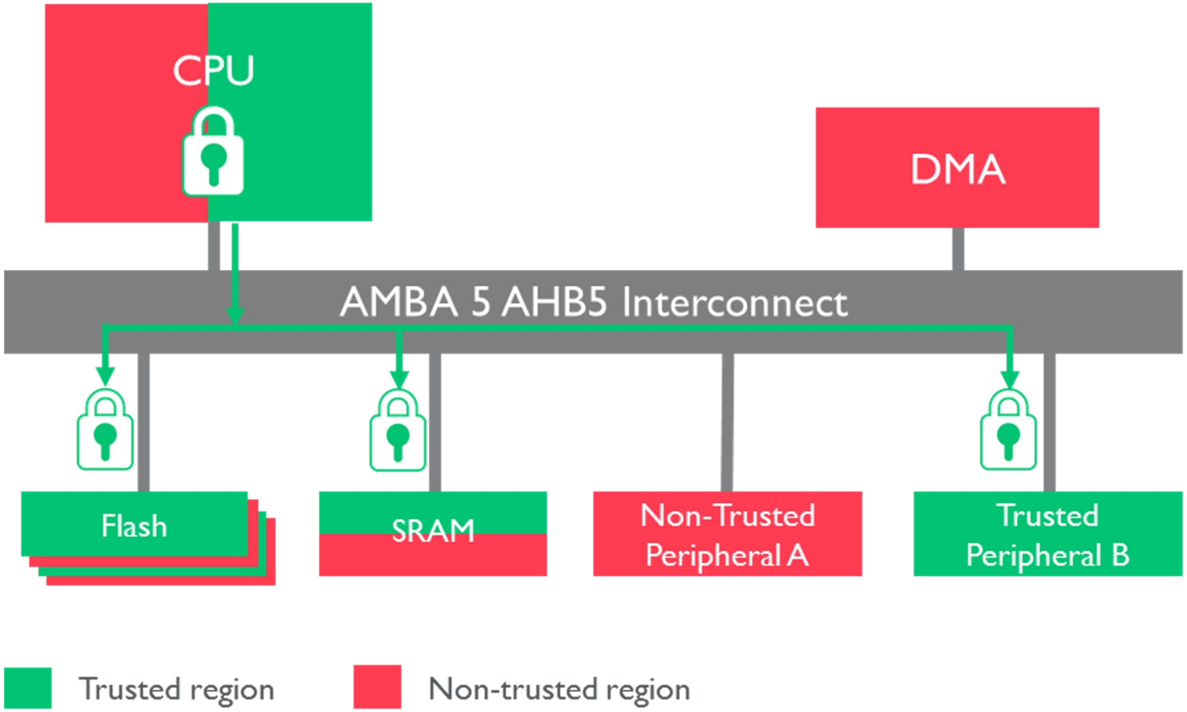
We address these challenges in **Tarnhelm**, which *transparently* executes individual code components in TrustZone and guarantees *code confidentiality* through isolation, without sacrificing overall system security.

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



Picture reused from arm.com

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.

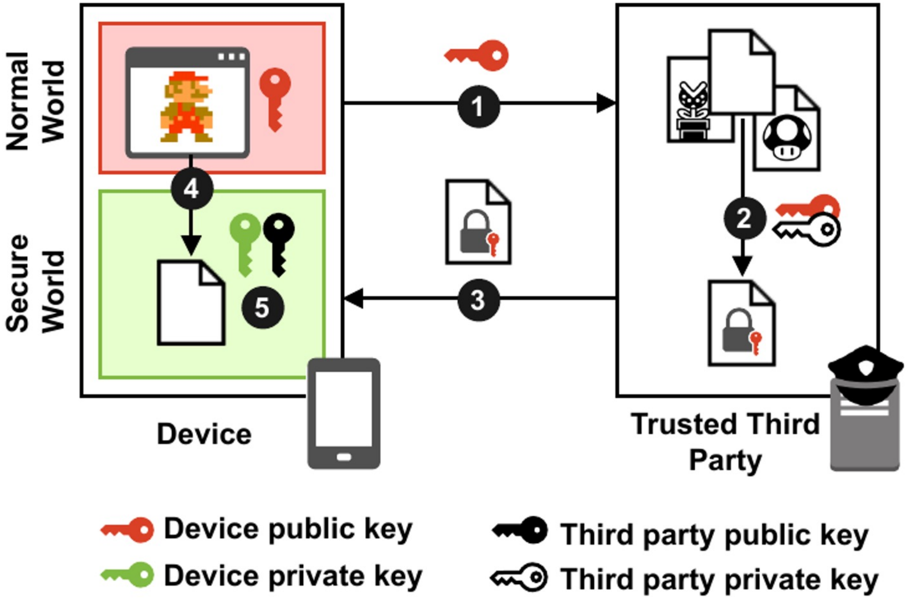
Design Goals

Design Goals

- Code confidentiality
- Transparent forwarding
- Transparent integration
- Limited attack surface
- Minimal overhead

Approach

Deployment

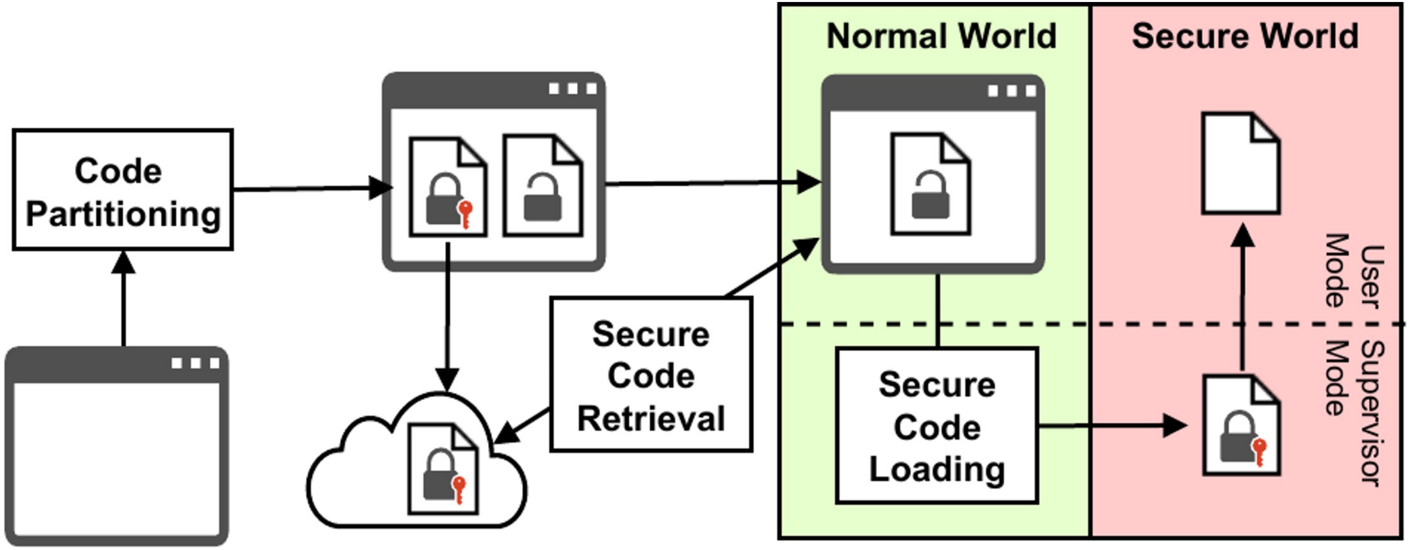


Normal World
Secure World
1
2
3
4
5
Device
Trusted Third Party
Device public key
Device private key
Third party public key
Third party private key

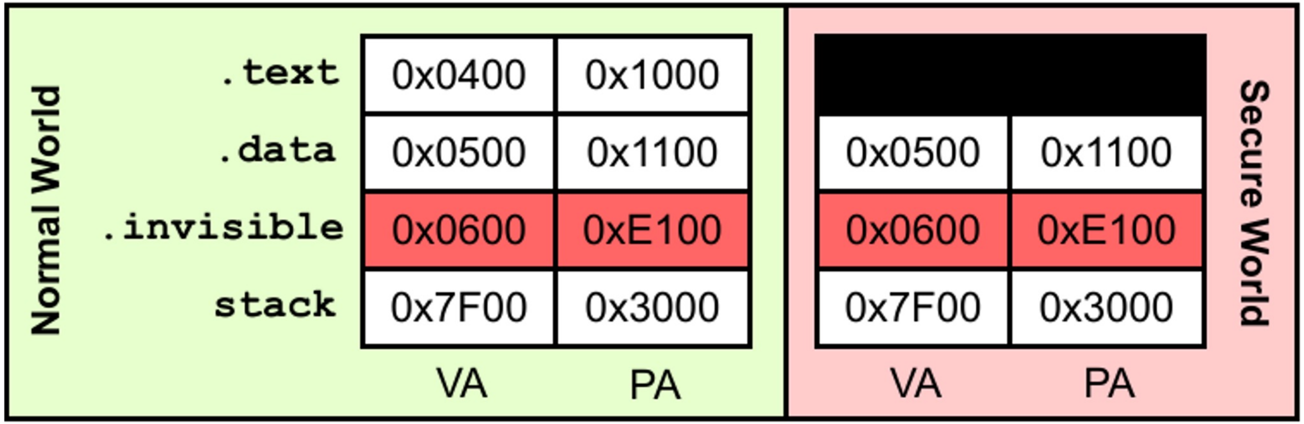
Code Partitioning

```
#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);
    ...
}
```

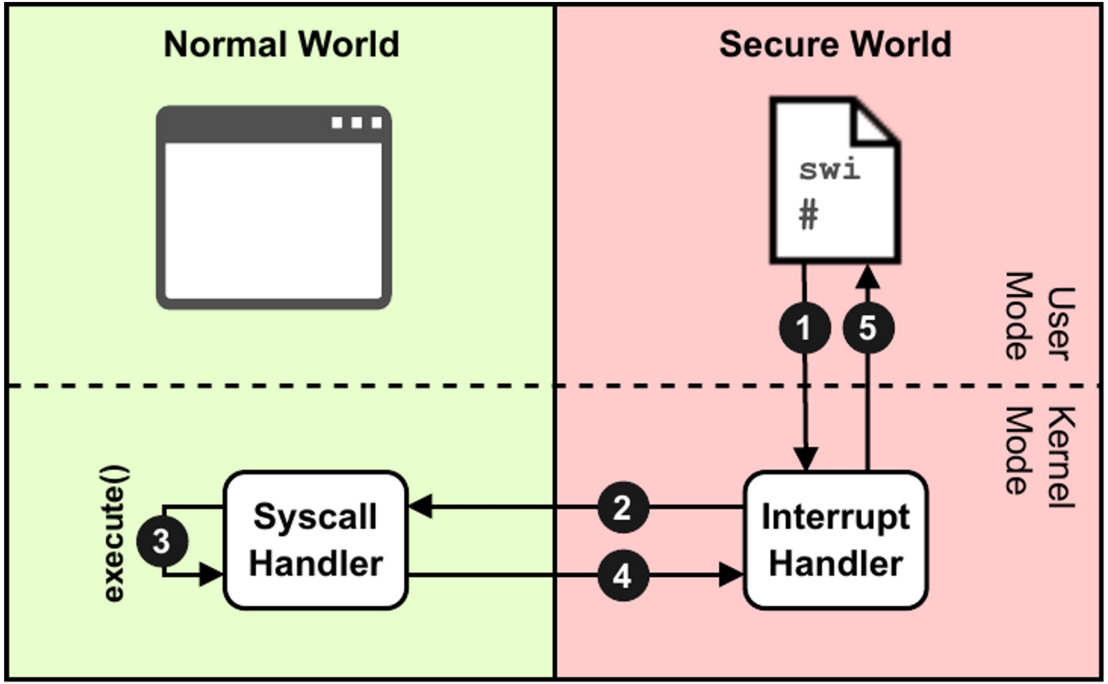
Secure Code Retrieval and Loading



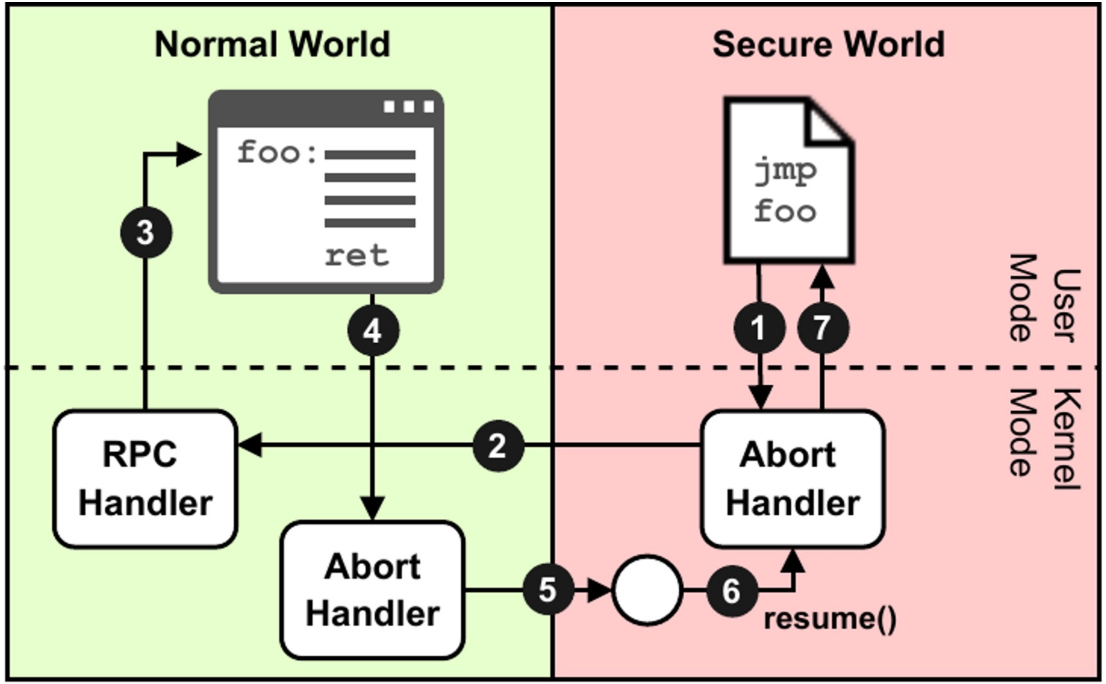
Memory Management



System Call Forwarding



Transparent World Switch



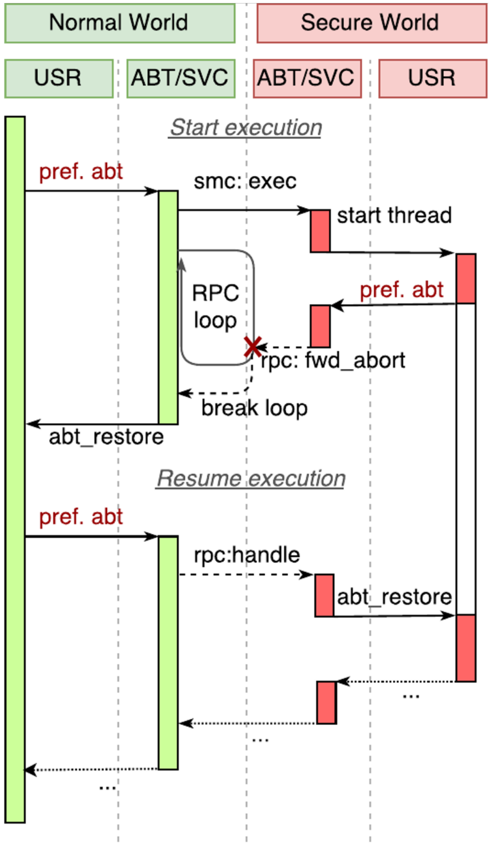
Implementation

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

↓From/To→		Untrusted OS	Trusted OS
Untrusted OS	`ret`	N/A	Verify and pop the return address from the shadow stack
	`call`	N/A	Verify function entry point and push return address on the shadow stack
Trusted OS	`ret`	Pop shadow stack	Verify return location to be valid
	`call`	Push return address on the shadow stack	Verify function entry point for indirect calls

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.

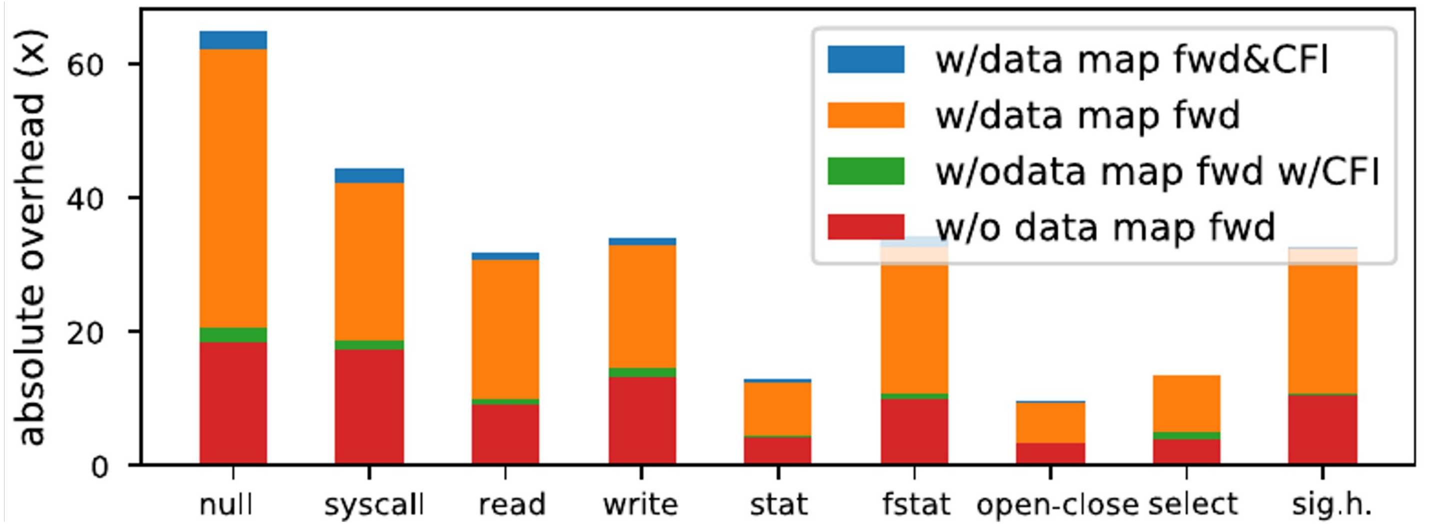
Microbenchmark of Tarnhelm's Individual Components

Component	Time
Invisible code initialization	0.316s
Invisible code cleanup	0.44ms
System call forwarding	116.88μs
Data mapping (secure world)	71μs
Data mapping (normal world)	231.337μs
IW-CFI indirect call (trusted OS)	0.111μs
IW-CFI return (trusted OS)	19.431μs

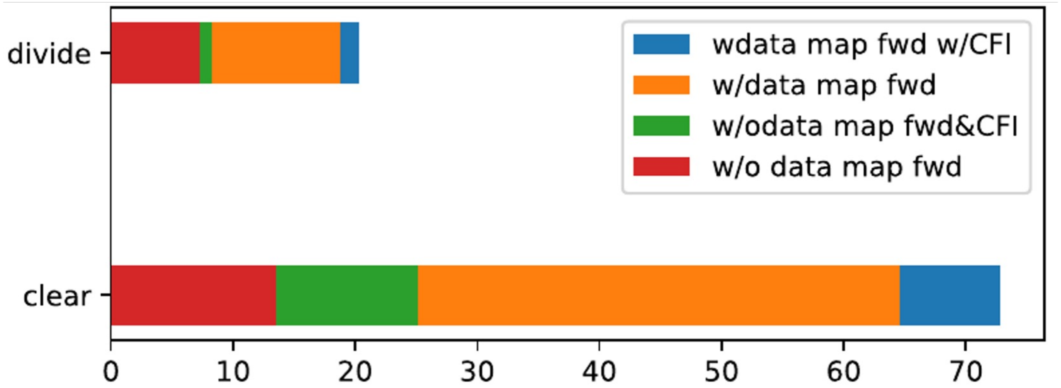
Overhead of the Transparent World Switch

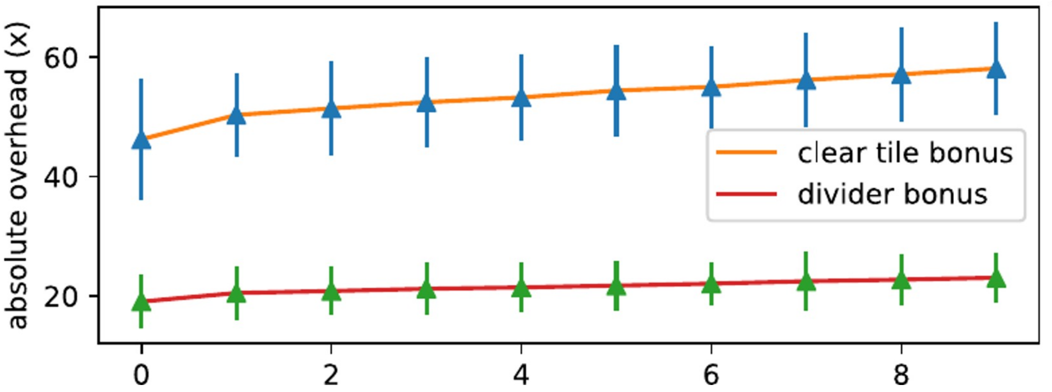
Direction	w/ DM+IWCFI	w/ DM fwd	w/o DM fwd
SW $\xrightarrow{\texttt{call}}$ NW $\xrightarrow{\texttt{ret}}$ SW	495.529μs	494.539μs	152.093μs
NW $\xrightarrow{\texttt{call}}$ SW $\xrightarrow{\texttt{ret}}$ NW	505.348μs	497.549μs	151.298μs
SW $\xrightarrow{\texttt{id-call}}$ NW $\xrightarrow{\texttt{ret}}$ SW	514.903μs	N/A	N/A

LMBench Results



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?