DICE*: A Formally Verified Implementation of DICE Measured Boot

Zhe Tao **zhetao@ucdavis.edu**
Aseem Rastogi aseemr@microsoft.com
Naman Gupta t-nag@microsoft.com
Kapil Vaswani kapilv@microsoft.com
Aditya V. Thakur avthakur@ucdavis.edu

30TH USENIX SECURITY SYMPOSIUM

UCDAVIS COMPUTER SCIENCE

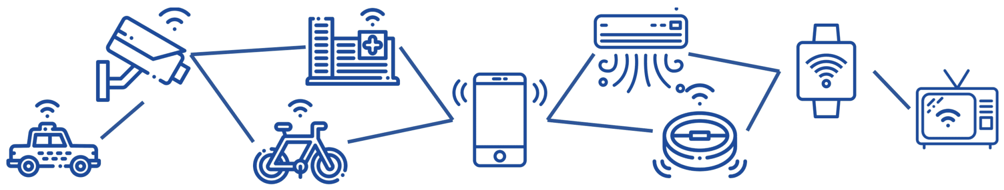
Microsoft Research

Establishing Trust in a Remote Device

Send sensitive data to ML accelerators on the cloud

Collect critical data from sensors in the field

Edge devices

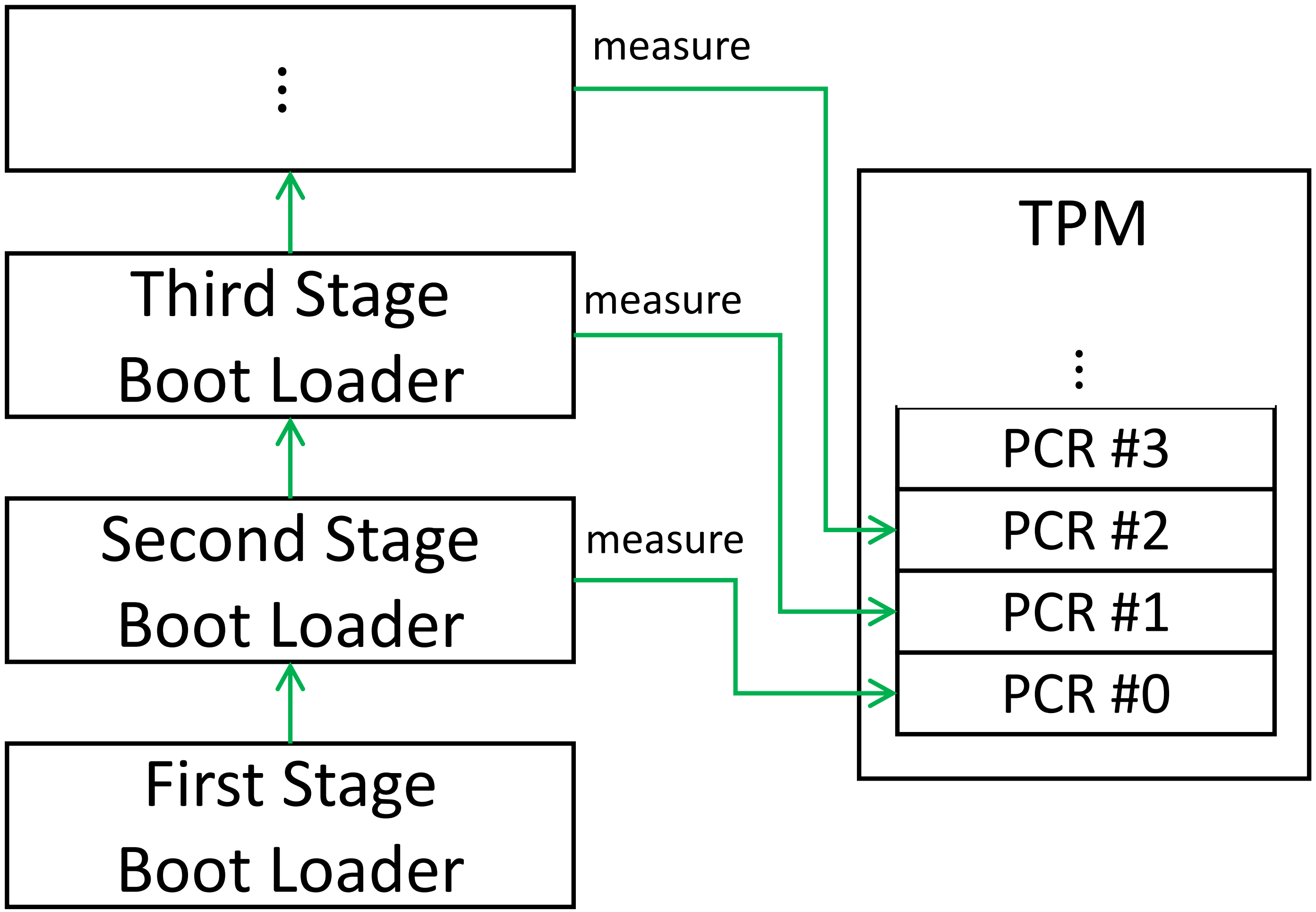


How do we verify that a device is running expected code?

Measured Boot

Prevents an attacker from loading unexpected code in boot by

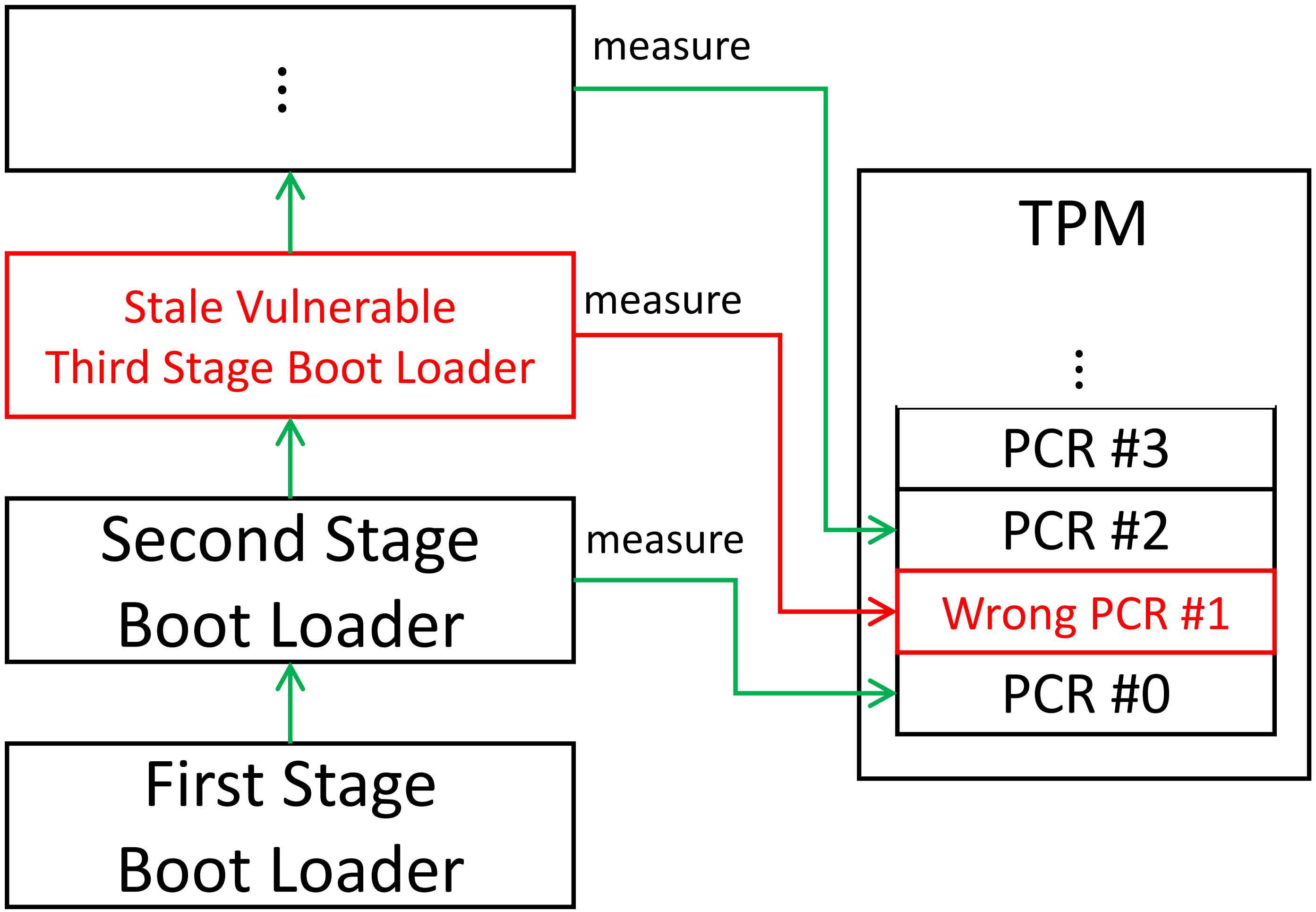
- measuring the boot sequence
- recording the measurements for later attestation



TPM	Trusted Platform Module
PCR	Platform Configuration Register

Measured Boot

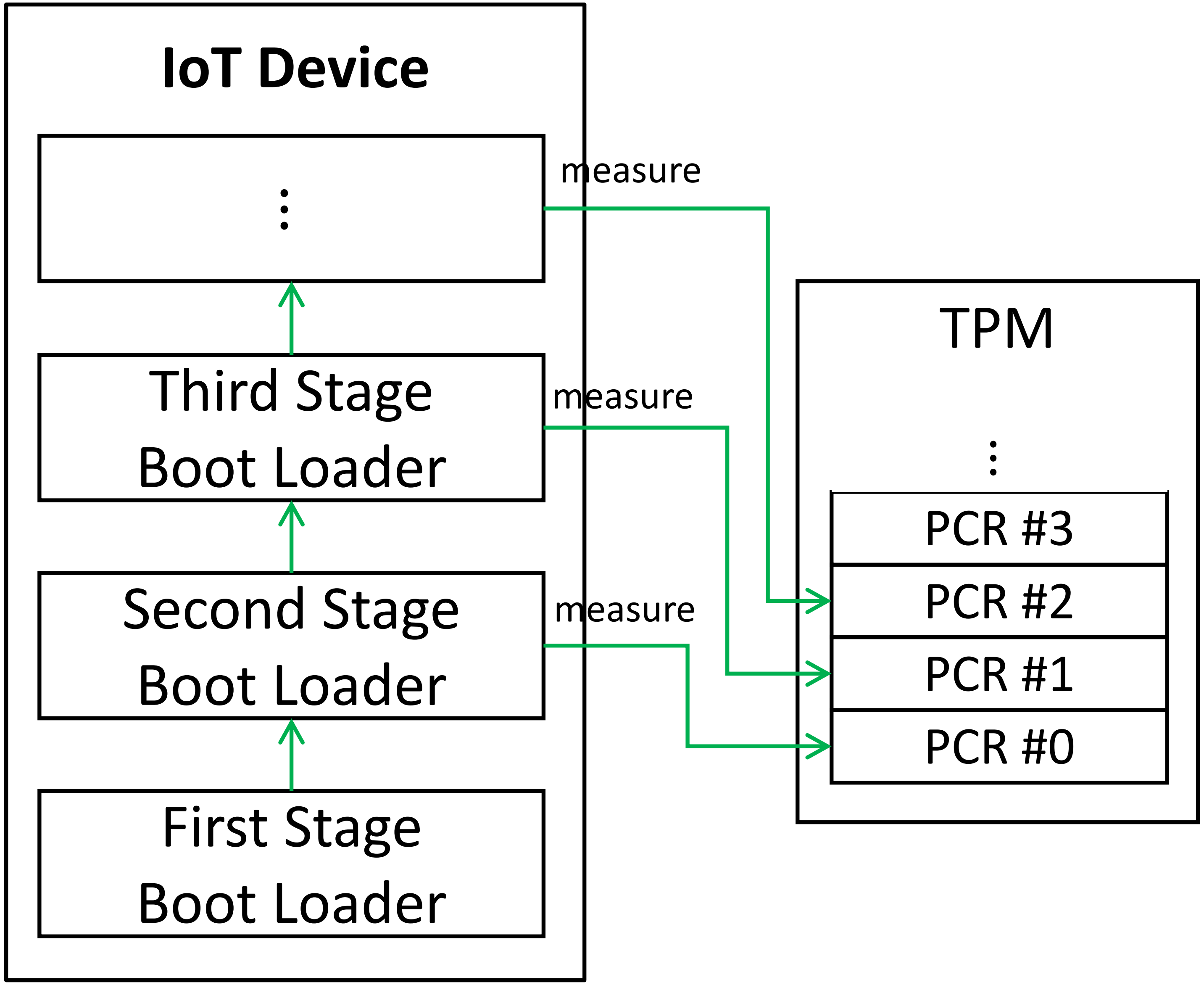
Unexpected code results in wrong measurement and fails attestation



TPM	Trusted Platform Module
PCR	Platform Configuration Register

Measured Boot

But traditional measured boot protocols are not applicable to IoT devices



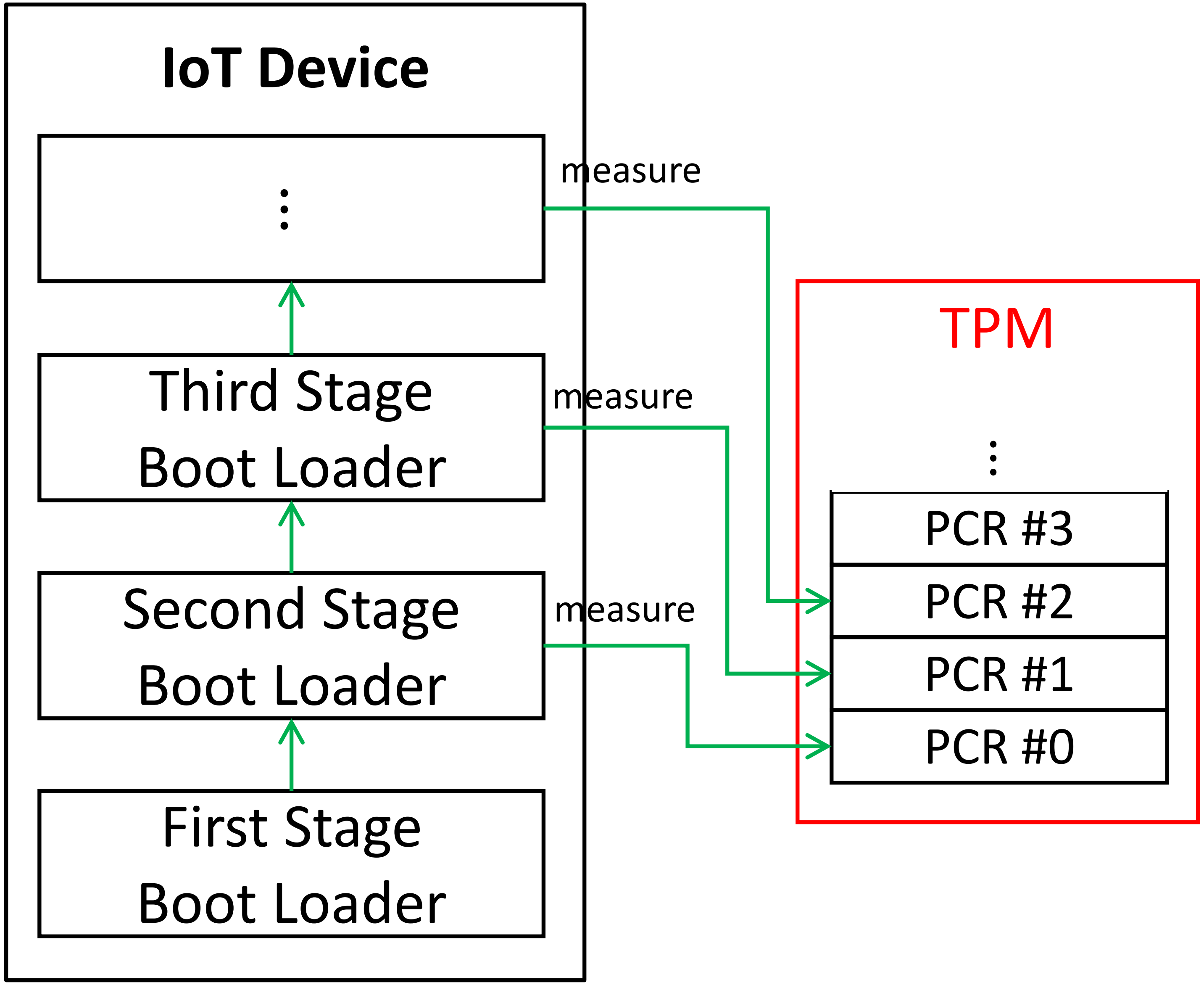
TPM Trusted Platform Module

PCR Platform Configuration Register

Measured Boot

But traditional measured boot protocols are not applicable to IoT devices

Because they require a dedicated chip like TPM, which is too expensive in terms of cost, power, or real estate



TPM Trusted Platform Module

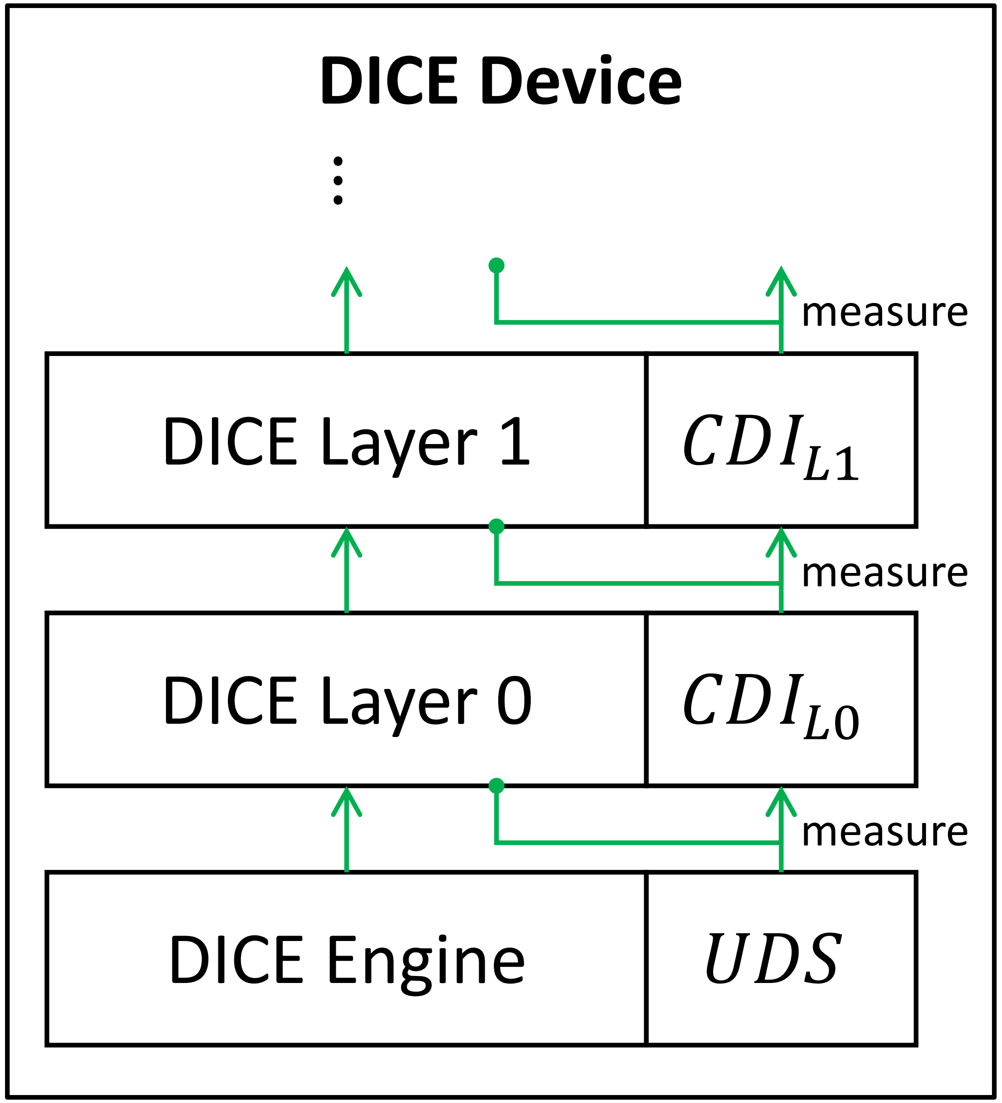
PCR Platform Configuration Register

DICE Measured Boot

Lightweight measured boot for IoT devices proposed by **Trusted Computing Group**.

DICE is becoming important. TCG members like Microsoft, STMicro, Microchip, Micron, NXP, etc. are behind its effort.

DICE is general for scenarios beyond IoT devices, like servers.



https://trustedcomputinggroup.org/work-groups/dice-architectures/

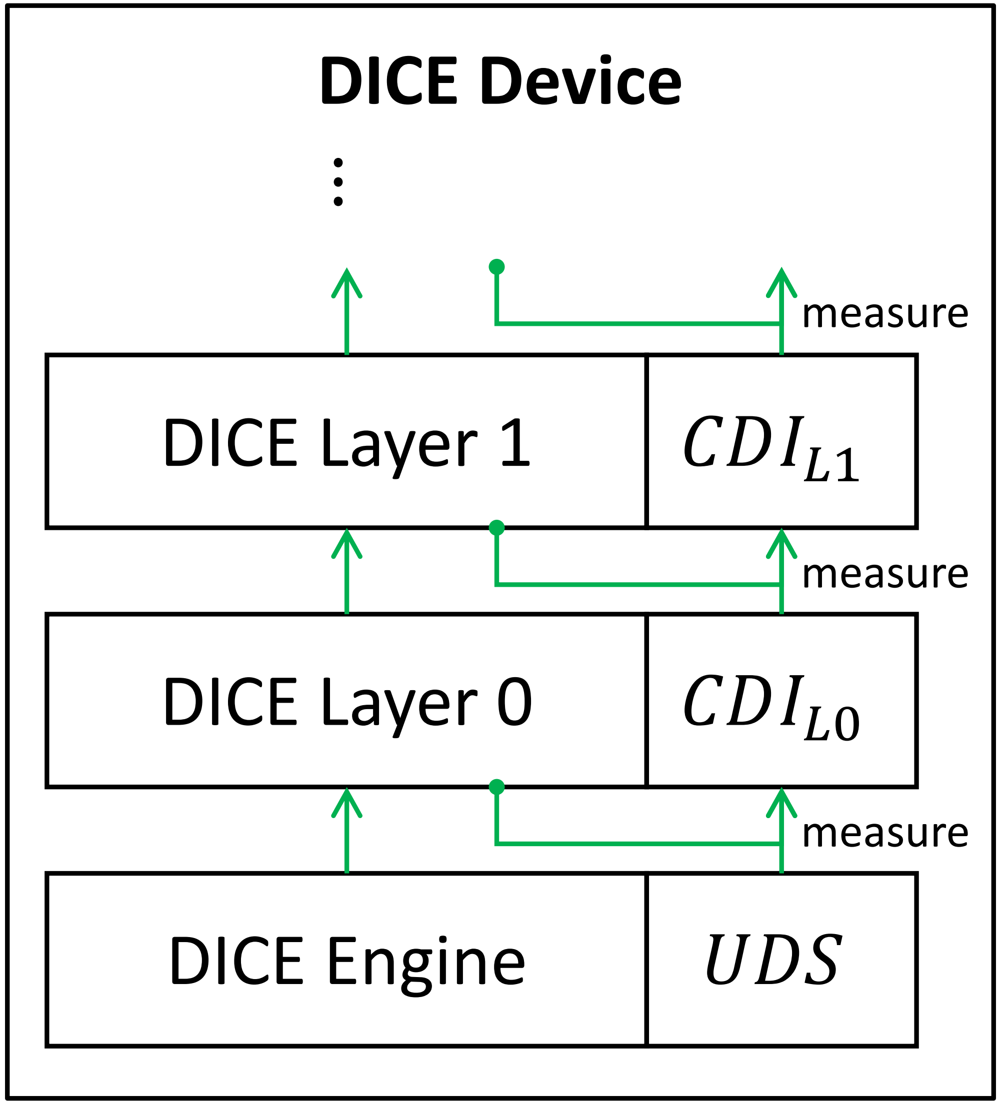
UDS Unique Device Secret

CDI Compound Device Identifier

DICE Measured Boot

DICE implicitly captures TCB as secrets (CDI) derived during boot

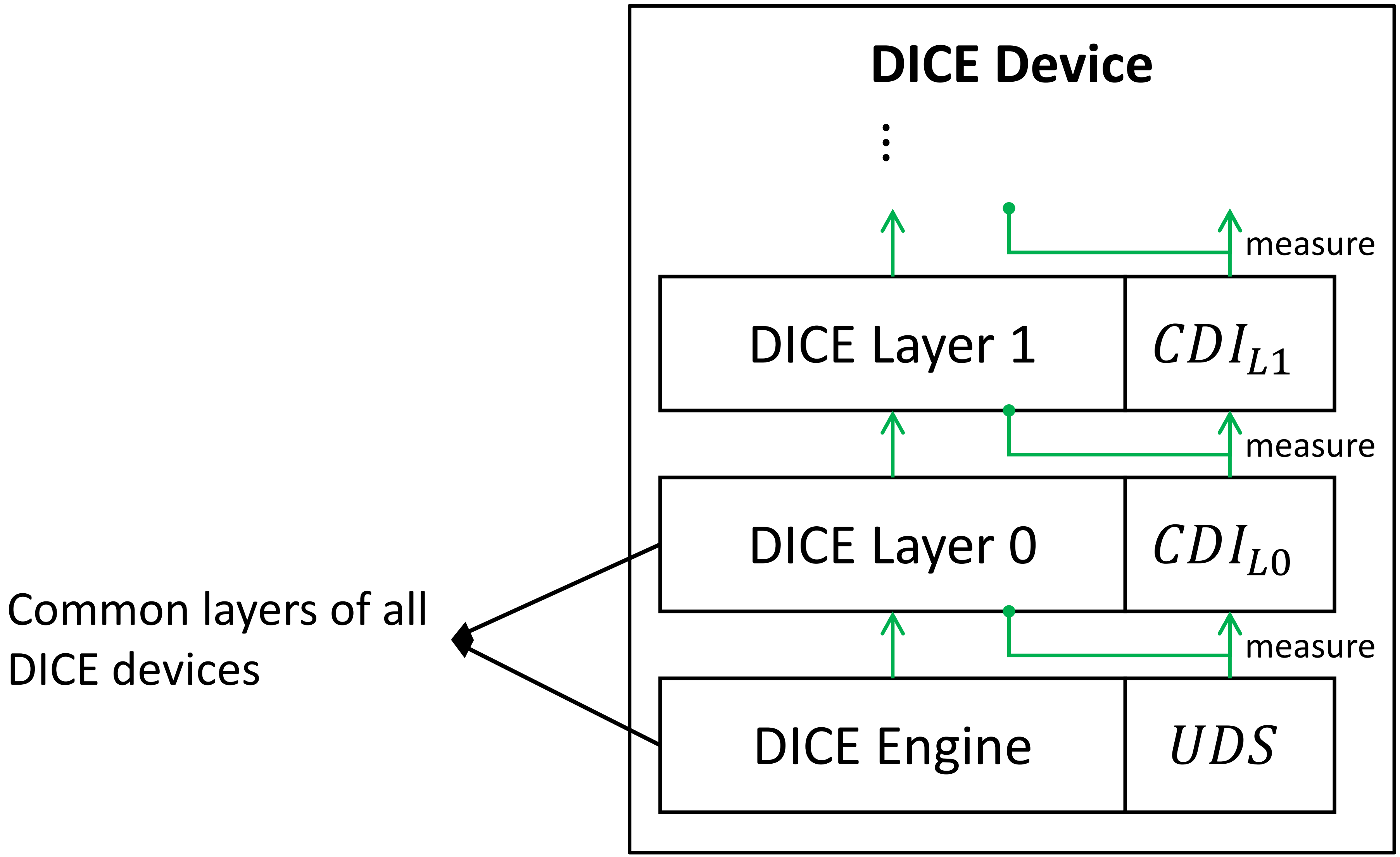
Layered structure:
each layer extends the TCB by measuring the upper layer and deriving a new CDI.



UDS Unique Device Secret

CDI Compound Device Identifier

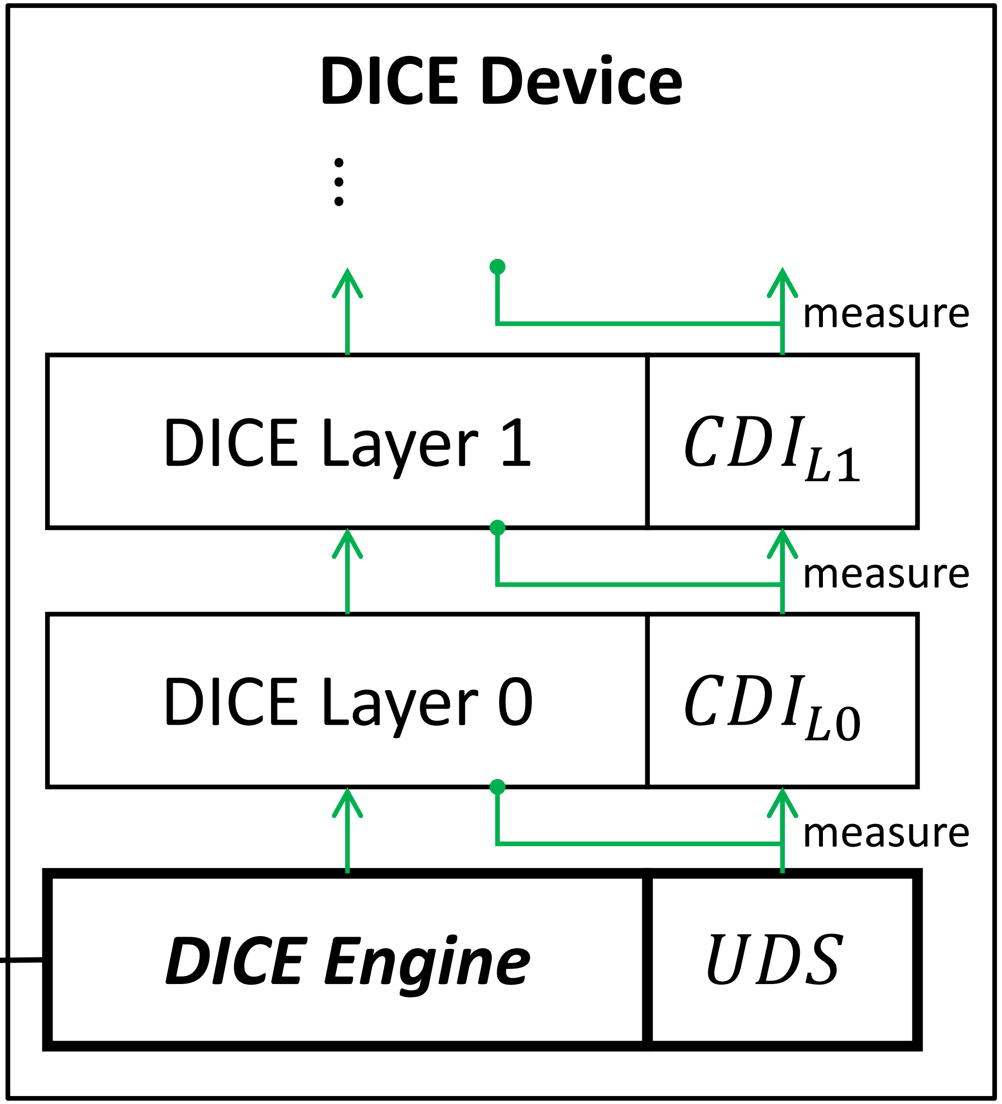
DICE Measured Boot



UDS Unique Device Secret

CDI Compound Device Identifier

DICE Measured Boot

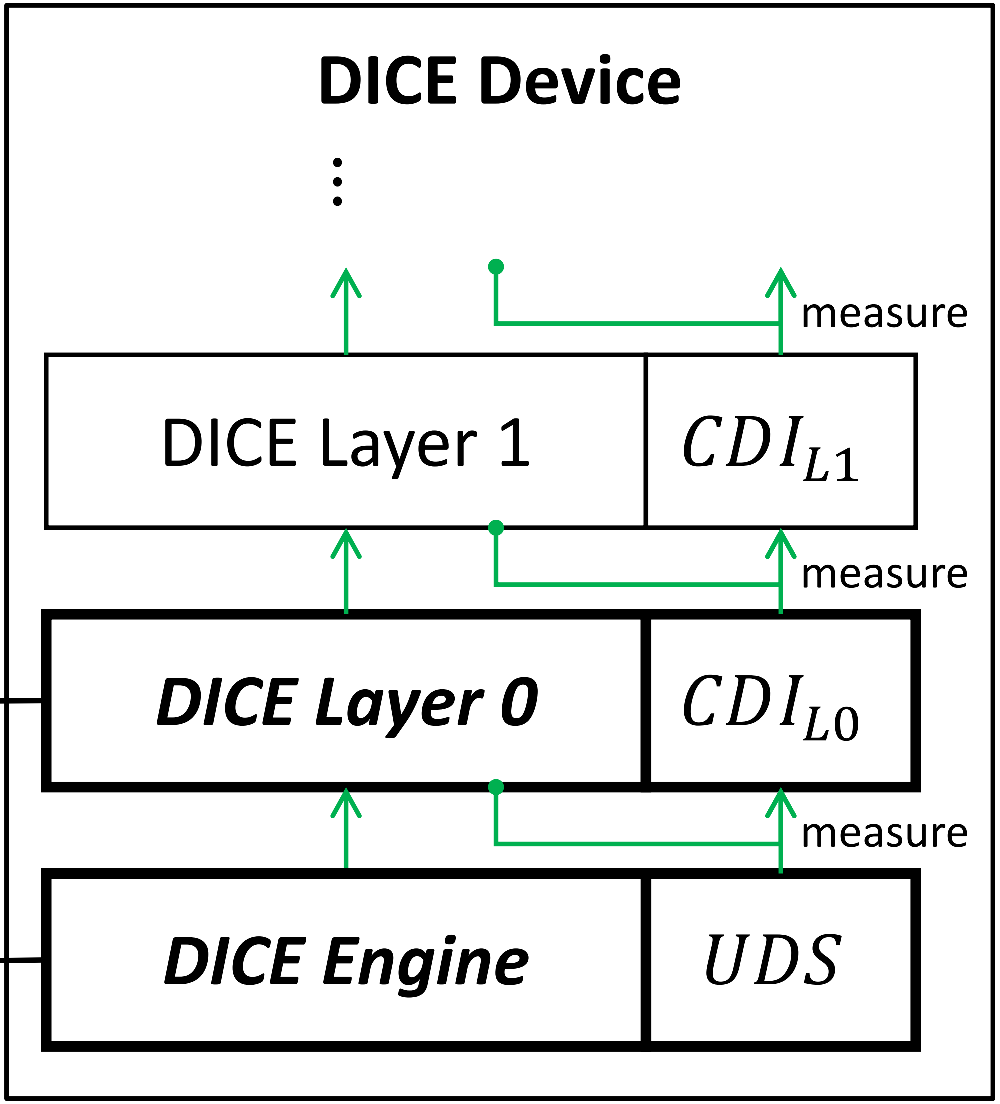


Minimal code to derive the first CDI and protects UDS

UDS Unique Device Secret

CDI Compound Device Identifier

DICE Measured Boot



Derives a life-time key and generates certificates

Minimal code to derive the first CDI and protects UDS

UDS Unique Device Secret

CDI Compound Device Identifier

Need for Verified DICE Implementation

DICE implementation is hard to get right because of key derivation, hashes, signatures and X.509 certificates — complex piece of code

National Security Agency | Cybersecurity Advisory

Patch Critical Cryptographic Vulnerability in Microsoft Windows Clients and Servers

Summary

NSA has discovered a critical vulnerability (CVE-2020-0601) affecting Microsoft Windows®[1] cryptographic functionality
The certificate validation vulnerability allows an attacker to undermine how Windows verifies cry
enable remote code execution. The vulnerability affects Windows 10 and Windows Server 2016
applications that rely on Windows for trust functionality. Exploitation of the vulnerability allows a
network connections and deliver executable code while appearing as legitimately trusted entitie
validation of trust may be impacted include:

- HTTPS connections
- Signed files and emails
- Signed executable code launched as user-mode processes

The vulnerability places Windows endpoints at risk to a broad range of exploitation vectors. NS

TPM-Fail

TPM meets Timing and Lattice Attacks

CVE-2016-2108 Detail

Current Description

The ASN.1 implementation in OpenSSL before 1.0.1o and 1.0.2 before 1.0.2c allows remote attackers to execute arbitrary code or cause a denial of service (buffer underflow and memory corruption) via an ANY field in crafted serialized data, aka the "negative zero" issue.

Severity CVSS Version 3.x | CVSS Version 2.0

CVSS 3.x Severity and Metrics:

Base Score: 9.8 CRITICAL **Vector:** CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

vailable information to associate vector strings and CVSS scores. We also display any CVSS information provided within the

Critical crypto bug leaves Linux, hundreds of apps open to eavesdropping

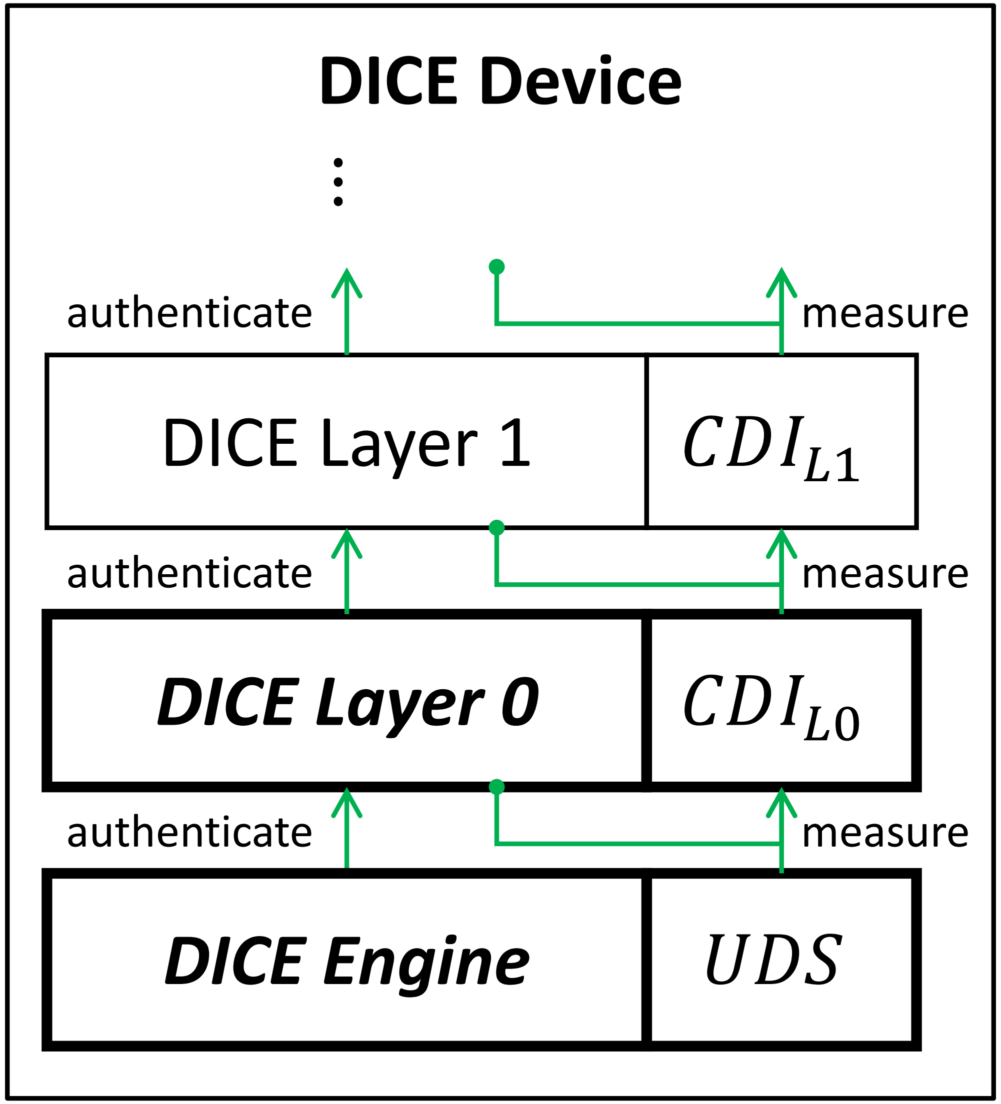
This GnuTLS bug is worse than the big Apple "goto fail" bug patched last week.

Need for Verified DICE Implementation

DICE is hard to get right because of the complex code and libraries

And bugs like memory errors, misuse of secrets, malleability attacks on X.509, side-channels may leak secrets allowing impersonation attack

But patching the first two layers is either impossible or extremely expensive



UDS Unique Device Secret

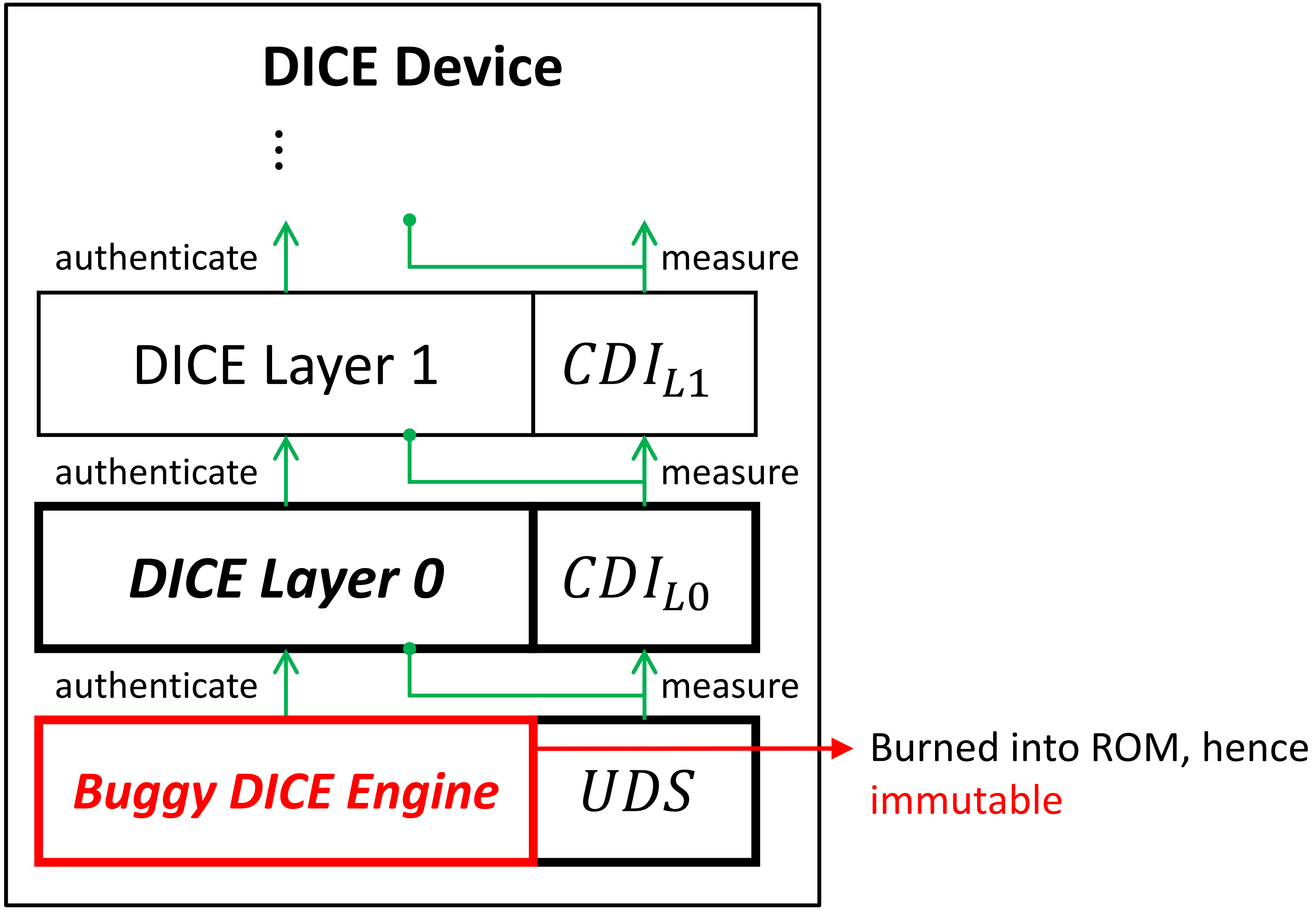
CDI Compound Device Identifier

Need for Verified DICE Implementation

DICE is hard to get right because of the complex code and libraries

And bugs like memory errors, misuse of secrets, malleability attacks on X.509, side-channels may leak secrets allowing impersonation attack

But patching the first two layers is either impossible or extremely expensive



UDS Unique Device Secret

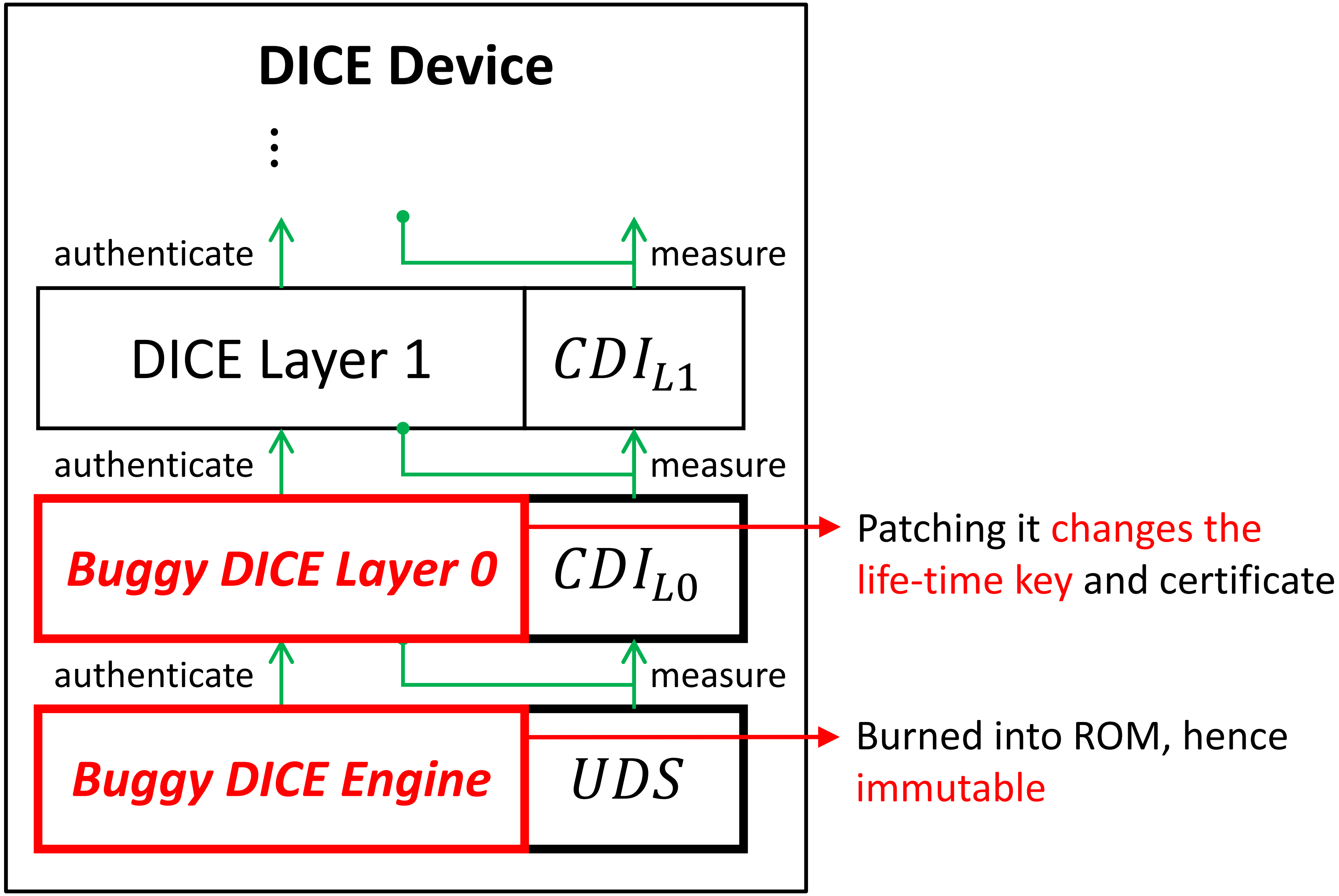
CDI Compound Device Identifier

Need for Verified DICE Implementation

DICE is hard to get right because of the complex code and libraries

And bugs like memory errors, misuse of secrets, malleability attacks on X.509, side-channels may leak secrets allowing impersonation attack

But patching the first two layers is either impossible or extremely expensive



UDS Unique Device Secret

CDI Compound Device Identifier

DICE*: A Formally Verified DICE implementation

- Formally specify DICE specification
- Present a verified DICE Engine with a platform-agnostic interface
  - Users can focus on analyzing the platform-specific components
- Present a verified DICE Layer 0
  - Including a verified library for a subset of X.509 which can be extended and reused
- Generate verified C implementation and evaluate it on STM32H753ZI
  - Comparable to unverified hand-written code in terms of boot time and binary size
- Available at https://github.com/verified-HRoT/dice-star

Verified Properties

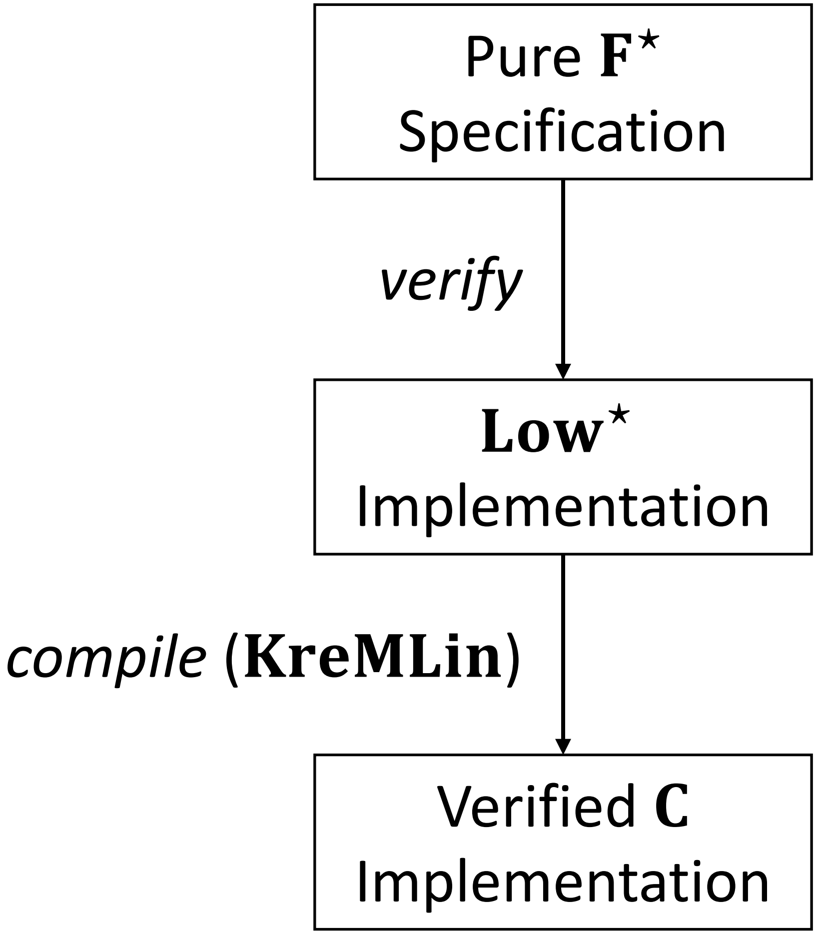
- Functional correctness
  - Secrets, keys and certificates are derived as per specification
- Memory safety
  - Buffer overflows, use-after-free, no null dereferences, dangling pointers, etc.
- Confidentiality
  - No secret leakage via outputs, memory, etc.
- Side-channel resistance
  - Free of certain timing- and cache-based side channels
- X.509 certificate security
  - No malleability attacks

Verification Toolchain

- $\mathbf{F}^\star$:
  - Functional language with effects
  - Dependent type
  - Semi-automated proof via SMT solvers

- $\mathbf{Low}^\star$: a shallow embedding of $\mathbf{C}$ in $\mathbf{F}^\star$
  - $\mathbf{C}$-like memory model
  - First Order
  - $\mathbf{C}$-compatible types

- $\mathbf{KreMLin}$: a $\mathbf{Low}^\star$-$\mathbf{C}$ compiler



DICE* Engine

DICE Engine

DICE* Engine uses a platform agnostic interface for UDS in different platforms

DICE* Engine models HMAC and Hash from **HACL***

Functional Specification of DICE Engine

$$\boxed{CDI_{L0}} = \boxed{HMAC}\left(\boxed{UDS}, \boxed{Hash}\left(\boxed{L0}\right)\right)$$

Platform-Agnostic Interface

Provides APIs
- Read UDS
- Latch UDS
- Erase stack

HACL* : a verified cryptographic library in **F***
- Memory Safe
- Functionally Correct
- Cryptographically Secure
- Side-Channel Resistant

Enforces the following behavior of **DICE*** Engine
- Cannot read UDS after latching UDS
- Must latch UDS before erasing stack
- Must erase stack before returning

UDS Unique Device Secret

CDI Compound Device Identifier

Verifying $\mathbf{DICE}^{\star}$ Engine: Top-Level Spec

Only uses stack

```
let dice_main ()
  : Stack dice_return_code
    (requires λ h → uds_is_enabled h)
    (ensures λ h0 r h1 → (¬ (uds_is_enabled h1)) ∧ stack_is_erased h1 ∧
      all_heap_buffers_except_cdi_and_ghost_state_remain_same h0 h1 ∧
      (r == DICE_SUCCESS ⟺ (l0_image_is_valid (st ()).l0 h1 ∧ cdi_functional_correctness (st ()) h1)))
```

UDS	Unique Device Secret
CDI	Compound Device Identifier

Verifying **DICE*** Engine: Top-Level Spec

```
let dice_main ()
  : Stack dice_return_code
    (requires λ h → uds_is_enabled h)
    (ensures λ h₀ r h₁ → (¬ (uds_is_enabled h₁)) ∧ stack_is_erased h₁ ∧
      all_heap_buffers_except_cdi_and_ghost_state_remain_same h₀ h₁ ∧
      (r == DICE_SUCCESS ⟺ (l0_image_is_valid (st ()).l₀ h₁ ∧ cdi_functional_correctness (st ()) h₁)))
```

Access to UDS is enabled.

Access to UDS is disabled after execution

Stack is erased

UDS Unique Device Secret

CDI Compound Device Identifier

Verifying **DICE*** Engine: Top-Level Spec

```
let dice_main ()
  : Stack dice_return_code
    (requires λ h → uds_is_enabled h)
    (ensures λ h₀ r h₁ → (¬ (uds_is_enabled h₁)) ∧ stack_is_erased h₁ ∧
      all_heap_buffers_except_cdi_and_ghost_state_remain_same h₀ h₁ ∧
      (r == DICE_SUCCESS ⟺ (l0_image_is_valid (st ()).l₀ h₁ ∧ cdi_functional_correctness (st ()) h₁)))
```

Returns SUCCESS iff $L0$ is authenticated and CDI_{L0} is $HMAC(UDS, Hash(L0))$

UDS Unique Device Secret

CDI Compound Device Identifier

Verifying **DICE*** Engine: Top-Level Spec

```
let dice_main ()
  : Stack dice_return_code
    (requires λ h → uds_is_enabled h)
    (ensures λ h0 r h1 → (¬ (uds_is_enabled h1)) ∧ stack_is_erased h1 ∧
      all_heap_buffers_except_cdi_and_ghost_state_remain_same h0 h1 ∧
      (r == DICE_SUCCESS ⟺ (l0_image_is_valid (st ()).l0 h1 ∧ cdi_functional_correctness (st ()) h1)))
```

Low* allows us to specify the following properties about memory:

- All ***heap*** buffers,
- which were ***alive*** at the initial state h_0,
- and are ***disjoint*** with the CDI buffer,
- are still ***alive*** at the final state h_1
- and are ***not modified.***

UDS Unique Device Secret

CDI Compound Device Identifier

Verifying **DICE*** Engine: Side-Channel Resistance

- **DICE*** follows the secret independent coding discipline by reusing the **secret integer model** from **HACL***
- **HACL*** defines secrets as abstract, constant-time integers which
  - can not be used as array indexes
  - can not be branched on because no Boolean comparison operators for them

$\mathbf{DICE}^{\star}$ Layer 0

Derive Asymmetric Key Pairs

$$DeviceID_{pub}, DeviceID_{priv} = KDF(CDI_{L0})$$

$$AliasKey_{pub}, AliasKey_{priv} = KDF(CDI_{L0}, L1\ )$$

Generate Certificates

$$CSR_{DeviceID} = Sign(CreateCSR\left(DeviceID_{pub}\right), DeviceID_{priv})$$

$$Crt_{AliasKey} \ \ = Sign(CreateCrt\ \left(AliasKey_{pub}\right), DeviceID_{priv})$$

KDF	Key Derivation Function
CSR	Certificate Signing Request
Crt	Certificate

DICE* Layer 0

Generate Certificates

$$CSR_{DeviceID} = Sign(\boxed{CreateCSR}(DeviceID_{pub}), DeviceID_{priv})$$

$$Crt_{AliasKey} = Sign(\boxed{CreateCrt}(AliasKey_{pub}), DeviceID_{priv})$$

X.509 Certificate Generation Library

provides verified serializer primitives and combinators for

- (Most of) ASN.1 constructs
- (A fragment of) X.509 messages

KDF	Key Derivation Function
CSR	Certificate Signing Request
Crt	Certificate

$\mathbf{DICE}^{\star}$ Layer 0

Derive Asymmetric Key Pairs

$$DeviceID_{pub}, DeviceID_{priv} = \boxed{KDF}(CDI_{L0})$$

$$AliasKey_{pub}, AliasKey_{priv} = \boxed{KDF}(CDI_{L0}, L_1)$$

Generate Certificates

$$CSR_{DeviceID} = \boxed{Sign}(CreateCSR(DeviceID_{pub}), DeviceID_{priv})$$

$$Crt_{AliasKey} = \boxed{Sign}(CreateCrt\,(AliasKey_{pub}), DeviceID_{priv})$$

$\mathbf{HACL}^{\star}$

KDF	Key Derivation Function
CSR	Certificate Signing Request
Crt	Certificate

Verified X.509 Certificate Generation Library

- We reuse the secure parser and serializer model from **LowParse** for specification
- We verify properties such as our serializers are injective

$$\forall m_1, m_2. s(m_1) = s(m_2) \Rightarrow m_1 = m_2$$

Verified X.509 Certificate Generation Library

- X.509 certificates are encoded into ASN.1

Verified X.509 Certificate Generation Library

- X.509 certificates are encoded into ASN.1 Tag-

$\boxed{Tag}$

Verified X.509 Certificate Generation Library

- X.509 certificates are encoded into ASN.1 Tag-Length-

Tag	$Length$

Verified X.509 Certificate Generation Library

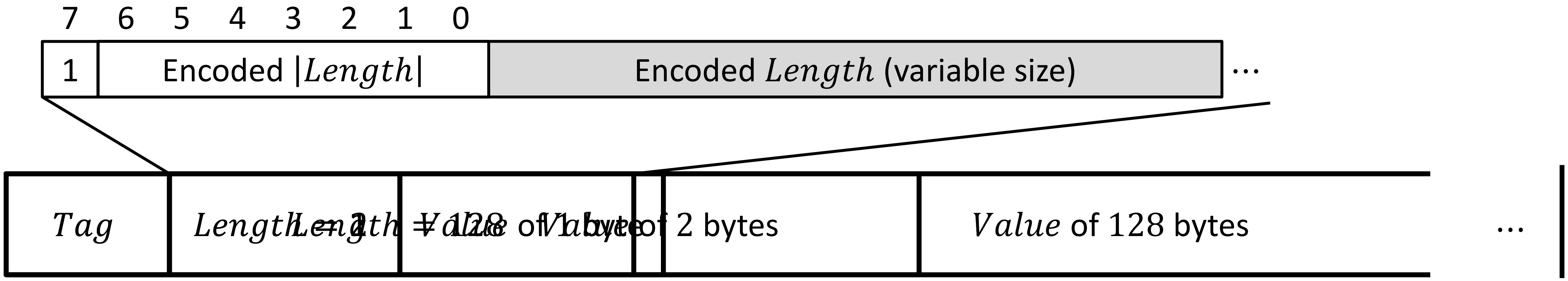
- X.509 certificates are encoded into ASN.1 Tag-Length-Value (TLV) format

Tag	$Length$	$Value$

Verified X.509 Certificate Generation Library

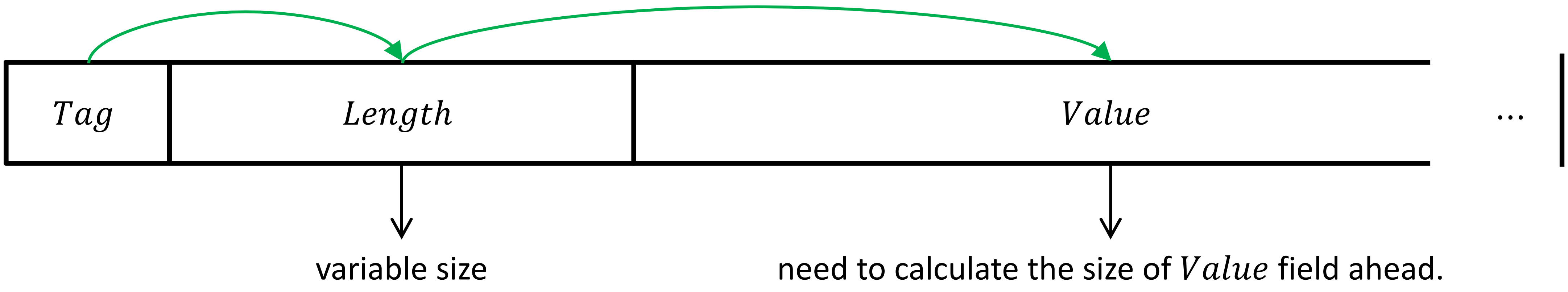
- X.509 certificates are encoded into ASN.1 Tag-Length-Value (TLV) format
  - where the length field specifies the size of the value field
- But the length field is also variable size!

When $Length \geq 128$



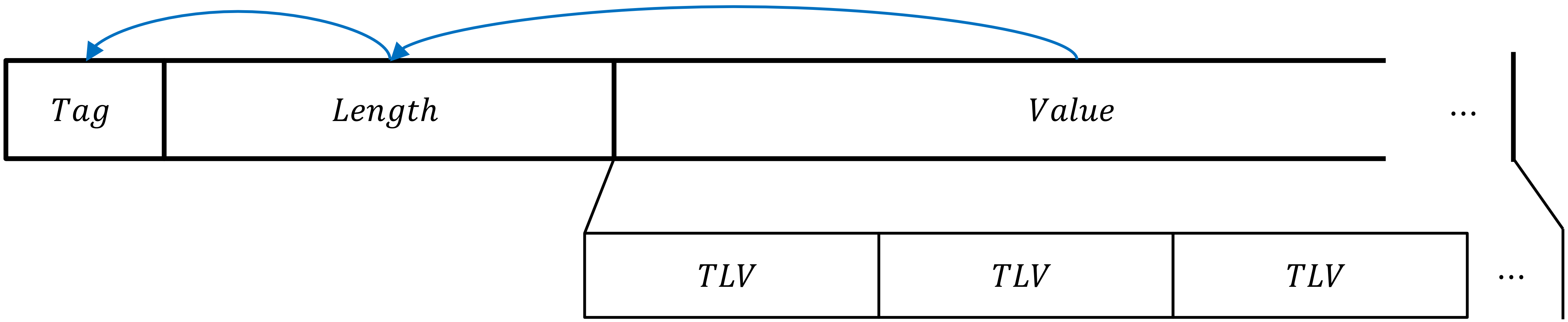
Verified X.509 Certificate Generation Library

- The low-level forward serializer from **LowParse** needs to calculate the size of value field ahead.
- Hence needs multiple passes to serialize an ASN.1 message, which is inefficient.



Verified X.509 Certificate Generation Library

- We implement a verified low-level backward serializer, which serializes an ASN.1 message in one pass
  - even in the presence of nested TLV messages.



DICE* Implementation

DICE* Layer 0

7,677 **F*** LoC
5,051 **C** LoC

DICE* Engine

533 **F*** LoC
205 **C** LoC

Platform-Agnostic Interface

X.509 Certificate Generation Library

16,564 **F*** LoC

~25k lines of **F*** code and proof
~5K lines of generated **C** code

DICE* Implementation

```c
dice_return_code dice_main()
{
  HWState_state s = st();
  bool b = authenticate_l0_image(s.l0);
  dice_return_code r;
  if (b)
  {
    KRML_CHECK_SIZE(sizeof (uint8_t), uds_len);
    uint8_t uds[uds_len];
    memset(uds, 0U, uds_len * sizeof (uint8_t));
    read_uds(uds);
    uint8_t uds_digest[32U];
    memset(uds_digest, 0U, (uint32_t)32U * sizeof (uint8_t));
    uint8_t l0_digest[32U];
    memset(l0_digest, 0U, (uint32_t)32U * sizeof (uint8_t));
    Hacl_Hash_SHA2_hash_256(uds, uds_len, uds_digest);
    Hacl_Hash_SHA2_hash_256(s.l0.l0_binary, s.l0.l0_binary_size, l0_digest);
    Hacl_HMAC_compute_sha2_256(s.cdi, uds_digest, (uint32_t)32U, l0_digest, (uint32_t)32U);
    zeroize(uds_len, uds);
    r = DICE_SUCCESS;
  }
  else
  {
    r = DICE_ERROR;
  }
  disable_uds();
  platform_zeroize_stack();
  return r;
}
```

https://github.com/verified-HRoT/dice-star/tree/main/dist

Evaluation

- We show that the C implementation generated from $\mathbf{DICE}^{\star}$ is comparable to the unverified hand-written one

Evaluation

- We show that the C implementation generated from $\mathbf{DICE}^{\star}$ is comparable to the unverified hand-written one in terms of both **boot time**

Layer	**Boot time (ms)**	
	Unverified DICE	$\mathbf{DICE}^{\star}$
DICE Engine	786	689
DICE Layer 0	313	208

Evaluation

- We show that the C implementation generated from $\mathbf{DICE}^{\star}$ is comparable to the unverified hand-written one in terms of both **boot time** and **binary size**.

Layer	Boot time (ms)		Size (KB)	
	Unverified DICE	$\mathbf{DICE}^{\star}$	Unverified DICE	$\mathbf{DICE}^{\star}$
DICE Engine	786	689	72	68
DICE Layer 0	313	208	92	92

DICE*: A Formally Verified DICE implementation

DICE* Layer 0

X.509 Certificate Generation Library

DICE* Engine

Platform-Agnostic Interface

https://github.com/verified-HRoT/dice-star

Thank you!

Zhe Tao **zhetao@ucdavis.edu**
Aseem Rastogi aseemr@microsoft.com
Naman Gupta t-nag@microsoft.com
Kapil Vaswani kapilv@microsoft.com
Aditya V. Thakur avthakur@ucdavis.edu

UC DAVIS COMPUTER SCIENCE
Microsoft Research