DICEx*: 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
Establishing Trust in a Remote Device

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

Send sensitive data to ML accelerators on the cloud

Collect critical data from sensors in the field
Measured Boot

Prevents an attacker from loading unexpected code in boot by
• measuring the boot sequence
• recording the measurements for later attestation

TPM

<table>
<thead>
<tr>
<th>PCR #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCR #2</td>
</tr>
<tr>
<td>PCR #1</td>
</tr>
<tr>
<td>PCR #0</td>
</tr>
</tbody>
</table>

TPM  Trusted Platform Module
PCR  Platform Configuration Register
Measured Boot

Unexpected code results in wrong measurement and fails attestation
Measured Boot

But traditional measured boot protocols are not applicable to IoT devices.
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.
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 \(CDI\).
DICE Measured Boot

Common layers of all DICE devices

DICE Device

\[ \vdots \]

DICE Layer 1  \[ CDI_{L1} \]

DICE Layer 0  \[ CDI_{L0} \]

DICE Engine  \[ UDS \]

UDS  Unique Device Secret

CDI  Compound Device Identifier
DICE Measured Boot

Minimal code to derive the first $CDI$ and protects $UDS$
DICE Measured Boot

DICE Device

DICE Layer 0

DICE Layer 1

DICE Engine

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
Patch Critical Cryptographic Vulnerability in Microsoft Windows Clients and Servers

Summary

NSA has discovered a critical vulnerability (CVE-2020-0601) affecting Microsoft Windows™ cryptography functions. The certificate validation vulnerability allows an attacker to undermine how Windows verifies certificates, enabling 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 entity 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. NSA recommends immediate patching.

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 3.0: 9.9 CRITICAL
CVSS Version 2.0: 10.0 CRITICAL

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

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.

**Need for Verified DICE Implementation**
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.

DICE Device

\[\begin{array}{c|c}
\text{authenticate} & \text{measure} \\
\hline
\text{DICE Layer 1} & CDI_{L1} \\
\text{authenticate} & \text{measure} \\
\text{Buggy DICE Layer 0} & CDI_{L0} \\
\text{authenticate} & \text{measure} \\
\text{Buggy DICE Engine} & UDS \\
\end{array}\]

\textit{Patching it changes the life-time key and certificate.}

\textit{Burned into ROM, hence immutable.}

\textit{UDS} Unique Device Secret
\textit{CDI} Compound Device Identifier

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

• **F***:
  • Functional language with effects
  • Dependent type
  • Semi-automated proof via SMT solvers

• **Low***: a shallow embedding of C in F***
  • C-like memory model
  • First Order
  • C-compatible types

• **KreMLin**: a Low***-C compiler
**DICE*** Engine

DICE Engine

**Functional Specification of DICE Engine**

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

Platform-Agnostic Interface

\[ CDI_{L0} = HMAC(UDS, Hash(L0)) \]

**HACL*** - a verified cryptographic library in **E***

- Memory Safe
- Functionality for stack operations
- Cryptographically Secure
- Side-Channel Resistant

Provides APIs
- Read UDS
- Latch UDS
- Erase stack

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 **DICE** Engine: Top-Level Spec

```plaintext
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 ↔ (λ_image_is_valid (st ())).λ₀ h₁ ∧ cdi_functional_correctness (st ()) h₁)))
```

**UDS**  Unique Device Secret

**CDI**    Compound Device Identifier
Verifying **DICE*** Engine: Top-Level Spec

```ml
let dice_main ()
: Stack dice_return_code
(requires ℼ ℓ ℓ₀ ℓ₁ → uds_is_enabled ℓ₁)
(ensures ℼ ℓ₀ ℓ₁ → (¬ (uds_is_enabled ℓ₁)) ∧ stack_is_erased ℓ₁ ∧
  all_heap_buffers_except_cdi_and_ghost_state_remain_same ℓ₀ ℓ₁ ∧
  (ℓ == DICE_SUCCESS ↔ (10_image_is_valid ℓ₀ ℓ₁ ∧ cdi_functional_correctness ℓ₀ ℓ₁)))
```

**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_heapBuffers_except_cdi_and_ghost_state_remain_same h₀ h₁ ∧
   r == DICE_SUCCESS ↔ \( L₀ \text{image_is_valid } (st ()).L₀ h₁ \) ∧
   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

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
**DICE* Layer 0**

**Derive Asymmetric Key Pairs**

\[
DeviceID_{pub}, DeviceID_{priv} = KDF(CDI_{L0})
\]

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

**Generate Certificates**

\[
CSR_{DeviceID} = Sign(CreateCSR(DeviceID_{pub}), DeviceID_{priv})
\]

\[
Crt_{AliasKey} = Sign(CreateCrt(AliasKey_{pub}), DeviceID_{priv})
\]

---

**Key Derivation Function**

**Certificate Signing Request**

**Certificate**
Generate Certificates

\[ CSR_{DeviceID} = \text{Sign}(\text{CreateCSR}(DeviceID_{pub}, DeviceID_{priv})) \]

\[ Crt_{AliasKey} = \text{Sign}(\text{CreateCrt}(AliasKey_{pub}, DeviceID_{priv})) \]

provides verified serializer primitives and combinators for
- (Most of) ASN.1 constructs
- (A fragment of) X.509 messages

\[ KDF \quad \text{Key Derivation Function} \]
\[ CSR \quad \text{Certificate Signing Request} \]
\[ Crt \quad \text{Certificate} \]
**DICE* Layer 0**

**Derive Asymmetric Key Pairs**

\[
DeviceID_{pub}, DeviceID_{priv} = KDF(CDI_{L0})
\]

\[
AliasKey_{pub}, AliasKey_{priv} = KDF(CDI_{L0}, L_1)
\]

**Generate Certificates**

\[
CSR_{DeviceID} = \text{Sign}(\text{CreateCSR}(DeviceID_{pub}, DeviceID_{priv}))
\]

\[
Crt_{AliasKey} = \text{Sign}(\text{CreateCrt}(AliasKey_{pub}, DeviceID_{priv}))
\]
• 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-
Verified **X.509 Certificate Generation Library**

- X.509 certificates are encoded into ASN.1 Tag-Length-
Verified X.509 Certificate Generation Library

- X.509 certificates are encoded into ASN.1 Tag-Length-Value (TLV) format
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 $\text{Length} \geq 128$

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 1 | Encoded $|\text{Length}|$ | Encoded $\text{Length}$ (variable size) | ...

| Tag | $\text{Length}$ | Value of 1 byte | Value of 2 bytes | Value of 128 bytes | ...
|-----|-----------------|-----------------|------------------|-------------------|---|
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.

![Diagram](image)
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

**X.509 Certificate Generation Library**
- 16,564 F* LoC

Platform-Agnostic Interface

~25k lines of F* code and proof
~5K lines of generated C code
```c
void dice_return_code (uint8_t uds_len, uint8_t * uds, uint8_t uds_sizeof)
{
    HWState_state s = st();
    bool b = authenticate_10_image(s.10);
    dice_return_code r;
    if (b)
    {
        KRML_CHECK_SIZE((sizeof (uint8_t) * uds_sizeof), uds_len);
        uint8_t * uds = malloc((sizeof (uint8_t) * uds_sizeof));
        memset(u, 0, uds_sizeof);
        read_u (uds);
        uint8_t uds_digest[32u];
        memset(uds_digest, 0,(sizeof (uint8_t) * uds_sizeof));
        uint8_t 10_digest[32u];
        memset(10_digest, 0,(sizeof (uint8_t) * uds_sizeof));
        HAC.HMAC_SHA2_256(uds, uds_sizeof, uds_digest);
        HAC.HMAC.HMAC_SHA2_256(uds, uds_sizeof, 10_digest);
        HAC.HMAC.HMAC_SHA2_256(uds, uds_sizeof, 10_digest);
        zeronize(uds, uds_sizeof);
        r = DICE_SUCCESS;
    }
    else
    {
        r = DICE_ERROR;
    }
    disable_u (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 DICE* is comparable to the unverified hand-written one
Evaluation

- We show that the C implementation generated from DICE* is comparable to the unverified hand-written one in terms of both boot time.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Boot time (ms)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unverified DICE</td>
<td>DICE*</td>
<td></td>
</tr>
<tr>
<td>DICE Engine</td>
<td>786</td>
<td>689</td>
<td></td>
</tr>
<tr>
<td>DICE Layer 0</td>
<td>313</td>
<td>208</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

• We show that the C implementation generated from DICE* is comparable to the unverified hand-written one in terms of both boot time and binary size.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Boot time (ms)</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unverified DICE</td>
<td>DICE*</td>
</tr>
<tr>
<td>DICE Engine</td>
<td>786</td>
<td>689</td>
</tr>
<tr>
<td>DICE Layer 0</td>
<td>313</td>
<td>208</td>
</tr>
</tbody>
</table>
**DICE**: A Formally Verified DICE implementation

**DICE**

- **Layer 0**
- **Engine**
- **Platform-Agnostic Interface**

**X.509 Certificate Generation Library**

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

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