DICE*: A Formally Verified Implementation of DICE Measured Boot

Zhe Tao[*] Aseem Rastogi[†] Naman Gupta[†] Kapil Vaswani[†] Aditya V. Thakur[*]

[*]*University of California, Davis* [†]*Microsoft Research*

Abstract

Measured boot is an important class of boot protocols that ensure that each layer of firmware and software in a device's chain of trust is measured, and the measurements are reliably recorded for subsequent verification. This paper presents DICE*, a formal specification as well as a formally verified implementation of DICE, an industry standard measured boot protocol. DICE* is proved to be functionally correct, memory-safe, and resistant to timing- and cache-based side-channels. A key component of DICE* is a verified certificate creation library for a fragment of X.509. We have integrated DICE* into the boot firmware of an STM32H753ZI micro-controller. Our evaluation shows that using a fully verified implementation has minimal to no effect on the code size and boot time when compared to an existing unverified implementation.

1 Introduction

Security attacks during boot are arguably the most difficult to defend against because at this stage in a device's lifecycle, traditional defences such as firewalls and anti-viruses are not in place, and attacks are hard to detect. It is, therefore, not surprising that securing devices during boot continues to be an active area of investigation [23, 27, 44, 50, 64].

A common defence against boot attacks is *authenticated* or *secure boot* [13]. In this form of boot, the device ROM is provisioned with a public key, which is used to authenticate the next layer of firmware. This ensures that the device can only boot with firmware signed by an authorized entity (e.g. the device manufacturer).

While authenticated boot forms the first line of defence in many systems, it remains susceptible to many attacks [33, 41]. For example, authenticated boot does not prevent an attacker from booting the device with an older version of firmware with known vulnerabilities. To prevent such attacks, many systems deploy a stronger, more secure boot protocol known as *measured boot* [41, 61]. Measured boot ensures that every layer of firmware/software is measured before booting, and that the measurements are reliably recorded for future verification. For example, the measurements can be used to attest the device to a remote party (e.g. an attestation service), which can inspect the measurements and decide if the device is running an expected version of firmware before establishing trust in the device and provisioning secrets such as keys or certificates.

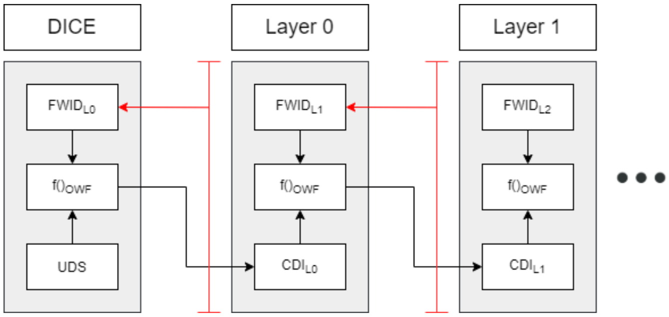

Figure 1: DICE Architecture

In many systems, measured boot is supported using a Trusted Platform Module (TPM) [28], a dedicated hardware chip attached to the host CPU. In a system with a TPM, each layer of firmware is configured to measure and record the hash of the next layer of firmware in the TPM. After boot, the TPM can generate a signed log of the firmware measurements using a unique signing key provisioned by the TPM manufacturer. This log can be verified externally to ascertain whether the device booted with expected firmware.

While suitable for some systems, there are many devices (e.g. low-cost IoT devices) where a dedicated TPM is too expensive in terms of cost, power, or real estate. To address the need for stronger security in such scenarios, researchers have recently proposed a new, measured boot architecture known as *Device Identifier Composition Engine* (DICE) [38], which is integrated on chip and requires significantly less hardware support.

In the DICE architecture (Figure 1), trust is anchored in a hardware component known as the *DICE engine*. Typically implemented in the ROM firmware, DICE engine is the first component to receive control when the device is powered-on. It has access to a *Unique Device Secret* (UDS) provisioned to each device during manufacturing. The engine transfers control to one or more layers of firmware/software, with the first layer known as *L0*. Before transferring control, it computes a *Compound Device Identifier* ($\mathrm{CDI}_{\mathrm{L0}}$) by measuring L0 and combining the measurement with the UDS using a One-Way hash Function (OWF). Every subsequent layer of the firmware measures its next layer and performs an analogous computation to obtain its CDI. Firmware layers may also derive additional secrets from their CDI. For example, L0 typically derives an asymmetric public/private key pair called the *DeviceID* from $\mathrm{CDI}_{\mathrm{L0}}$. A manufacturer-issued certificate for DeviceID can serve as the device's long term identity and can be used to validate the attestations originating from the device after deployment.

By incorporating the measurement of the next layers of firmware into CDI, DICE architecture ensures that the full Trusted Computing Base (TCB) of the device is implicitly captured in the secrets/keys derived during boot. An important consequence is that a change in the TCB (e.g. due to a firmware upgrade) automatically changes the keys derived during boot. Therefore, devices running with stale or compromised firmware cannot impersonate known good firmware.

Due to these security properties, minimal hardware requirements, simplicity, and low cost, DICE-based measured boot is being incorporated into an increasingly larger number of devices [8–11], and is being developed into a standard by Trusted Computing Group (TCG) [55, 56]. However, along side a standard, it is critical to develop a methodology that allows chip manufacturers to build and analyze robust, bug-free implementations of the standard because defects in these implementations can have serious implications, including attackers taking control of these devices. What is worse, fixing a defect in DICE engine or L0 layers is either impossible (if the layer is implemented in boot ROM), or extremely expensive because an update changes the device identity and invalidates the manufacturer issued certificates. Issuing new certificates for devices already deployed in the field may require decommissioning or recalling the affected devices, both of which can be expensive and/or laborious.

Building robust implementations of DICE is a challenging task for several reasons. Firstly, even though the DICE architecture is simple, its implementation contains complex cryptographic primitives such as public key derivation, signatures, and hashes, and generation of X.509 Certificate Signing Requests (CSR) and certificates in multiple layers of the firmware stack. Cryptographic and X.509 libraries are often written in a low-level unsafe language like C and are well-known for their security vulnerabilities and functional correctness bugs [1–6, 20]. Secondly, if the implementations are not careful operating on the secrets, the attackers may be able to infer them using side-channel leaks, e.g. timing, as in the TPM-Fail attack [42]. Finally, DICE implementations rely on hardware-specific security features to protect secrets and prevent tampering of code. These must be individually certified as part of any security analysis.

1.1 Our Contributions

In this paper, we present DICE*, the first formally-verified implementation of the standardized DICE engine layer [57] and L0 [58]. DICE* is proven to be memory-safe, functionally correct, secure, and resistant to the timing- and cache-based side-channel attacks. We implement DICE* in Low* [48], a shallow-embedding of a well-behaved subset of C inside the F* programming language and proof assistant [54]. Low* programs enjoy the full higher-order expressiveness of F* for specifications and proofs, while their first-order computational fragment can be extracted to efficient, readable, and portable C code using the KreMLin tool. For cryptographic primitives, DICE* uses HACL* [65], a formally verified cryptographic library written in Low*. For X.509 certificates, we extend the LowParse framework [49] and build a custom, verified X.509 certificate creation library for DICE. Concretely, we make the following contributions.

We show how DICE implementations can be refactored into platform-agnostic and platform-specific components that interact through a well-defined interface. This refactoring enables reuse of the platform-agnostic components across devices, thereby simplifying the security analysis (Section 4.1).

We formalize the DICE engine and L0 standards [57, 58] by designing their top-level (platform-agnostic) APIs with formal specifications (dice_main in Section 4.2 and l0_core in Section 5.3 resp.).

We provide a formally verified implementation of the platform-agnostic components in the DICE engine (Section 4.2) and L0 (Section 5.3) that is memory-safe, functionally correct, secure, and side-channel resistant. This verified implementation is applicable to all DICE devices, leaving the device manufacturers with a simpler task of analyzing just the platform-specific components.

We precisely specify (and verify) the outputs from each layer (CDI, keys, CSRs, and certificates), guaranteeing that there are no direct flows of secrets (e.g. UDS) to the outputs. Further, using the model of secrets as abstract types from Low*, DICE* also ensures that there are no secret-dependent branches or memory accesses, providing a constant-time implementation [17] that is resistant to the timing- and cache-based side channel attacks.

A key component of DICE* is a custom, verified X.509 certificate creation library (Section 5.2), implemented using the LowParse framework [49]. We extend LowParse with backward serializer support for serializing variable-length data. This extension is general and can be applied to any system that uses variable-length messages. The verified library that we have developed for (a subset of) ASN.1 and X.509 can be extended and applied to other applications, e.g. Public Key Infrastructure (PKI). We have laid the necessary groundwork by providing parser and serializer specifications, and low-level serializers for many of the basic types.

We evaluate DICE* by integrating it into the boot firmware of an STM32H753ZI micro-controller [11] and measuring the impact of the verified code on the firmware binary size (a critical metric for applicability to the low-cost devices) and boot time (Section 7). Our evaluation shows that using a fully verified implementation has minimal to no impact when compared to an unverified hand-written C implementation.

DICE* is publicly available at `https://github.com/verified-HRoT/dice-star`. DICE is a security-critical infrastructure component. By formally verifying it and producing a deployment-ready artifact, we hope that DICE* will serve as a robust baseline for the next generation of DICE implementations, thereby avoiding the expensive bug-finding and fixing cycles in the future.

The rest of this paper is structured as follows. Section 2 provides a background on DICE. Section 3 provides a high-level overview of our verification toolchain. Sections 4 and 5 focuses on the verification of DICE engine and L0 layers, respectively. Section 6 provides details of the DICE* implementation. Section 7 describes a DICE*-based implementation for the STM32H753ZI micro-controller, and compares this implementation with an unverified implementation. We review related work in Section 8 and conclude in Section 9.

2 Overview of DICE

This section describes the DICE architecture in more detail, discusses our threat model, verification goals, and TCB.

2.1 DICE Architecture

The DICE architecture is motivated by the need for a low cost measured boot protocol that can generate verifiable attestations capturing the entire hardware and software TCB of each device, and can be deployed on a large class of devices.

Figure 2 shows the dataflow in the simplest instance of the DICE architecture with three layers. The first layer is a hardware layer called the DICE engine [57], which receives control after device reset. This layer has access to the unique device secret (UDS) provisioned to the device during manufacturing. The DICE specification requires UDS to provide at least 256-bit cryptographic strength. The UDS must also be stored in read-only and latchable memory so that access to the UDS can be disabled and is restored only by a hardware reset. DICE engine performs the following sequence of operations:

1. **Authenticate L0 firmware.** First, the DICE engine loads the L0 firmware image into the RAM and authenticates it. One way of authenticating the image is to append the hash of the firmware image and a signature over this hash using a firmware signing key to the image, and provision the public firmware signing key to the device during manufacturing e.g. in e-fuses. The DICE engine can use this key to verify the signature, and check that the hash matches the hash of the image.

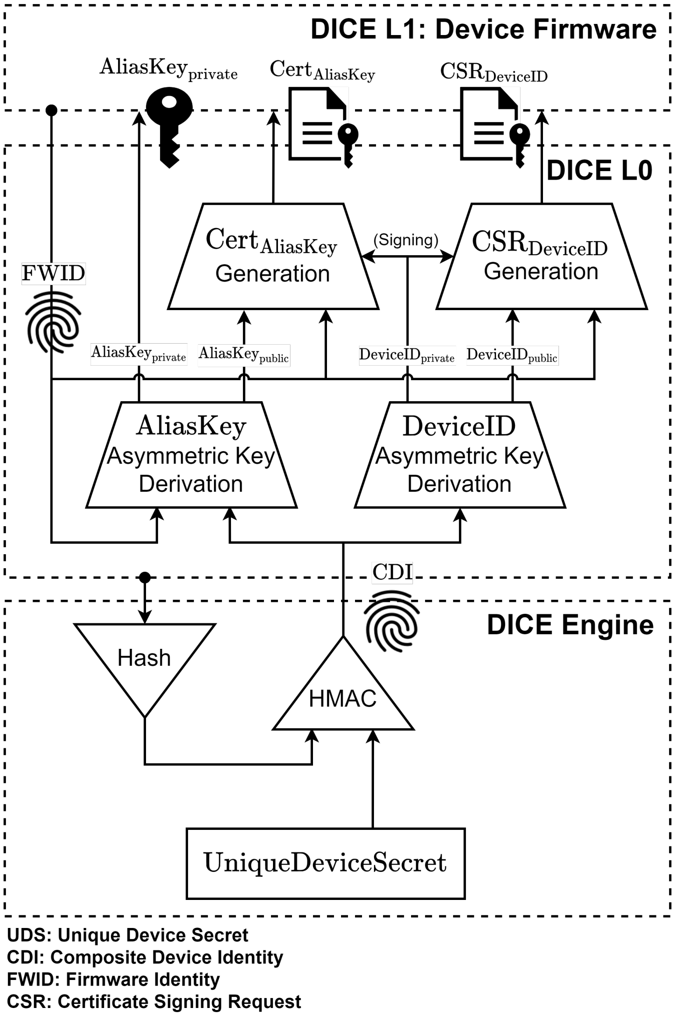
DICE L1: Device Firmware
AliasKey$_{\text{private}}$
Cert$_{\text{AliasKey}}$
CSR$_{\text{DeviceID}}$
DICE L0
Cert$_{\text{AliasKey}}$ Generation
(Signing)
CSR$_{\text{DeviceID}}$ Generation
FWID
AliasKey$_{\text{private}}$
AliasKey$_{\text{public}}$
DeviceID$_{\text{private}}$
DeviceID$_{\text{public}}$
AliasKey Asymmetric Key Derivation
DeviceID Asymmetric Key Derivation
CDI
DICE Engine
Hash
HMAC
UniqueDeviceSecret
UDS: Unique Device Secret
CDI: Composite Device Identity
FWID: Firmware Identity
CSR: Certificate Signing Request

Figure 2: DICE architecture with three layers of firmware.

2. **Derive CDI.** The DICE engine then derives a compound device identifier (CDI) from the UDS and the hash digest of the firmware image:

$$\mathrm{CDI}_{\mathrm{L0}} = \mathrm{HMAC}(\mathrm{UDS}, \mathrm{Hash}(\mathrm{L0})) \tag{1}$$

The DICE specification prescribes the use of the UDS as the HMAC key for the HMAC function, instead of a hash combining the UDS with the hash of L0. This derivation ensures that the derived CDI value has the same cryptographic strength as UDS (see NIST SP800-57, Part 1 [16]).

3. **Latch UDS.** The DICE engine disables access to the UDS using a hardware-specific latch mechanism, which

remains in place until the next reset. DICE engine also erases any copies of the UDS on the stack or in memory.

4. **Transfer to L0.** Finally, DICE engine passes the CDI value and control to the L0 firmware. To prevent Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks, it is crucial that the DICE engine jumps to the L0 firmware copy in the RAM from Step 1.

Together, these steps limit exposure of the UDS: access to the raw UDS value is restricted to only the DICE engine, and other firmware layers get access to the CDI derived from the UDS using a cryptographically secure one-way hash function.

2.2 Layer 0

Layer 0 (L0) is the layer of firmware that receives control from the DICE engine. Its main purpose is to derive an asymmetric device identity key (also known as DeviceID) from CDI using a cryptographically secure key derivation function (KDF):

$$\text{DeviceID}_{pub}, \text{DeviceID}_{priv} = \text{KDF}(\text{CDI}) \qquad (2)$$

If the KDF is cryptographically secure, i.e. injective and one-way, the derivation ensures that DeviceID uniquely identifies each device and the L0 firmware that the device is running. Furthermore, the public key DeviceID_{pub} does not reveal any information about CDI.

In most deployments, the L0 firmware (and consequently the device identity) is intended to remain unchanged throughout the lifetime of the device, unless there is a firmware corruption or an attempt to tamper. Therefore, the CDI value and DeviceID keys remain stable throughout the device lifetime.

The L0 layer is also responsible for generating X.509 Certificate Signing Requests (CSR) for DeviceID_{pub}. These CSRs are typically harvested in a trusted environment (e.g. during manufacturing), and processed by the manufacturer's PKI for DeviceID certificate issuance.

In addition to DeviceID, the L0 firmware generates an additional asymmetric key pair, known as the Alias Key, from CDI and the measurement of L1 (referred as FWID):

$$\text{AliasKey}_{pub}, \text{AliasKey}_{priv} = \text{KDF}(\text{CDI}, \text{FWID}) \qquad (3)$$

This key pair is unique for each combination of UDS, L0 firmware, and L1 firmware. It can be used by L1 for attestation and secure key exchange. L0 also issues an X.509 certificate for the alias key signed by DeviceID_{priv}. Therefore, any relying party can verify that the alias key was issued by a genuine device as long as they have access to a manufacturer issued DeviceID certificate for the device.

2.3 Threat Model

We focus on an adversary that has both remote and physical access to the device. Remotely, the attacker may try and exploit any vulnerability in the device firmware, and thereby obtain full control over execution including the ability to run arbitrary code. Physically, the adversary can observe or tamper with any of the device's I/O interfaces such as SPI, I2C, wi-fi, and any additional pins such as RESET and interrupts, as well as any persistent storage on the device e.g. flash memory. Finally, similar to HACL* [65], we assume that the adversary can observe the low-level runtime behavior such as branching and memory-access patterns.

Possible attacks. An attacker with these capabilities may exploit a buggy DICE implementation in several ways. A low-level memory error (such as a buffer overflow) or a simple bug in the implementation may leak secrets such as the UDS or the device private key into one of the outputs. Using a functional correctness bug in the X.509 certificate generation code, an attacker may load stale or malicious firmware on the device, while exploiting the bug to generate the certificate corresponding to a good firmware. Finally, if the implementation is not careful with the secrets, an attacker may be able to infer them by observing the branching behavior or memory-access patterns at runtime.

More sophisticated attacks such as exploiting speculative execution, fault injection, cold boot attacks, and use of electron microscopes to exfiltrate secrets are out of scope of this paper. In many simpler devices such as IoT devices, attacks based on speculative execution are not applicable because the CPUs do not use speculation. Attacks during manufacturing and in the supply chain such as leakage of secrets, device counterfeiting etc. are also out-of-scope.

2.4 Verification Goals

Our objective is to develop DICE implementations that guarantee that each device has a unique long-term identity and is capable of generating reliable assertions about its firmware even in the presence of an attacker with the capabilities described above. The verified implementation should satisfy the following properties:

Confidentiality. The DICE implementation should not leak any secrets or values derived from the secrets to the adversary. For instance, the UDS should only be accessible to the DICE engine, and the private DeviceID key should only be known to the L0 firmware.

Functional correctness. The DICE implementation should meet all functional requirements laid out in the DICE specification, including key derivation and certificate generation.

Memory safety. The DICE implementation should be free from low-level memory errors such as memory leaks, buffer overflows, null dereferences, and dangling pointers.

Side-channel resistance. At runtime, the sequence of instructions executed and memory access patterns should be independent of the secrets. Therefore, even an attacker who has access to the low-level branching and addresses of all memory accesses should not be able to distinguish between two runs that use two different values for secrets. In other words, the implementation should be resistant to timing- and cache-based side-channel attacks.

2.5 Trusted Computing Base

Our TCB includes the Low* toolchain, including the F* typechecker, Z3 SMT solver, and the KreMLin compiler. Low* verification guarantees, including side-channel resistance, extend only until the compiled C code. Beyond that, one may use a certified C compiler like CompCert [18] that preserves both the semantics and the constant-time property of the input C code, or use a more general compiler like gcc at the cost of adding it to the TCB. We trust the native, platform-specific implementation of the hardware functions that our DICE engine implementation relies on (Section 4.1), the bootloader, I/O and peripheral drivers (Section 7.1), as well as the native (one-line) implementation of declassification routine used to declassify public keys (Section 5.3). Finally, we assume that the manufacturer deploys a secure PKI infrastructure that issues certificates only to genuine devices.

3 Overview of the Toolchain

We use the Low* toolchain to develop DICE*. Low* has been used to verify, generate, and deploy low-level code such as cryptographic algorithms [47, 65] and parsers and serializers [49]. By developing DICE* also in Low*, we are able to integrate with these libraries at the specification level, thereby providing strong end-to-end guarantees. In this section, we provide a background of the toolchain.

3.1 F*

F* [54] is a dependently-typed functional programming language that allows programmers to do proofs about their programs—programmers write specifications as part of the types, and with the help of SMT-based automation provided by F*, prove that their program meet those specifications. As an example, the factorial function in F* can be given the type $\mathsf{int} \rightarrow \mathsf{int}$, as in other languages like OCaml, but it can also be given a more precise type $\mathsf{x}{:}\mathsf{int}\{\mathsf{x} \geq 0\} \rightarrow \mathsf{y}{:}\mathsf{int}\{\mathsf{y} \geq \mathsf{x}\}$. The type states that the function must be called with non-negative int arguments, and it returns int-typed results that are at least as large as their arguments (the type $\mathsf{x}{:}\mathsf{int}\{\mathsf{x} \geq 0\}$ is called a refinement type). F* type system is also *effectful*—the function types in F* capture the effects of the function body. $\mathsf{x}{:}\mathsf{t}_1 \rightarrow \mathsf{t}_2$ is a shorthand for $\mathsf{x}{:}\mathsf{t}_1 \rightarrow \mathsf{Tot}\ \mathsf{t}_2$, where Tot is the effect of pure, terminating computations. Note that we write the argument type as $\mathsf{x}{:}\mathsf{t}_1$ to emphasize that x may appear free in t_2. Computations that work with mutable state have ST effect, with types of the form $\mathsf{x}{:}\mathsf{t}_1 \rightarrow \mathsf{ST}\ \mathsf{t}_2\ \mathsf{req}\ \mathsf{ens}$. When F* verifies a function to have this type, the metatheory of F* guarantees that if the function is called with an argument of type t_1 and in a state that satisfies the precondition req, then the function either diverges, or returns a value of type t_2 and the final state satisfies the postcondition ens.

F* programs can be extracted to OCaml (or C if they are written in the Low* fragment (Section 3.2)); the extraction only outputs computationally relevant code, erasing all the proofs and specifications.

Erased types F* standard library provides a mechanism to define values and computations that can only be used in the specifications and do not have any computational relevance. In particular, the ghost version erased t of any type t is noninformative and extracted as unit. To use an erased value, one must use the reveal function $\mathsf{reveal}{:}\ \mathsf{erased}\ \mathsf{t} \rightarrow \mathsf{Ghost}\ \mathsf{t}$, that incurs the Ghost effect. Again, terms with Ghost effect are computationally irrelevant, and are erased during extraction.

3.2 Low*

Low* [48] is a restricted, first-order subset of F* that can be used to program and verify low-level applications. Low* exposes shallow-embedding of a well-behaved subset of C in F* in the form of a C-like memory model with stack and heap, and libraries for machine integers and mutable arrays. While the Low* computational code is restricted to be first-order, proofs and specifications are free to use the full expressiveness of F*. Verified Low* programs can be extracted to readable and idiomatic C code that is free of low-level memory errors (such as buffer overflows, use-after-free, null pointer dereferences) and enjoys the specifications proven in Low*.

3.3 HACL*

HACL* [65] is a cryptographic library written and verified in Low*. In addition to being free of low-level memory errors, HACL* algorithms are also proven functionally correct and side-channel resistant (in the program-counter security model [43]). Because our DICE engine and L0 specifications are written using the specifications exported by HACL* primitives, we explain them in more detail.

Functional correctness of HACL* primitives. To prove the functional correctness of a cryptographic algorithm, say the SHA256 hash algorithm, HACL* defines a formal specification written in the pure fragment of F* that has no side-effects and is guaranteed to terminate. The specification is written using functional sequences (instead of mutable C arrays), and is free to use mathematical integers and natural numbers, or any other high-level constructs that may not have a low-level C counterpart:

```
type sbyte = u8 (∗ the type for secret bytes ∗)
let sha256_spec (inp:seq sbyte{length inp ≤ 2^61 − 1})
  : lseq sbyte 32 = ...
```

In this code snippet, the spec function for SHA256 takes as argument a sequence of bytes with the refinement capturing the allowed maximum length of the input, and returns a sequence of bytes whose length is 32. Its body implements the SHA256 algorithm. This specification is extracted to OCaml and tested on standard test vectors, but is otherwise trusted.

HACL⋆ then defines the low-level implementation of the primitive in Low⋆, using mutable arrays and bounded integers libraries, and relates it to the pure specification in the postcondition; e.g.,

```
let sha256_impl (len:size_t) (inp:array sbyte len) (dst:array sbyte 32)
: Stack unit
  (requires λh →
    len ≤ 2^61 − 1 ∧ live m inp ∧ live m dst ∧ disjoint [inp; dst])
  (ensures λh0 () h1 → modifies dst h0 h1 ∧
    as_seq h1 dst == sha256_spec (as_seq h0 inp))
```

The Low⋆ array type array t len represents C-arrays with element type t and length len. Effect label Stack is a refinement of ST that additionally ensures that sha256_impl does not perform any heap allocations. The precondition, a predicate on the input memory h, requires that the input arrays are live (temporal memory safety), and constrains their lengths as required by the SHA-256 algorithm (spatial memory safety). It also requires that inp and dst arrays are disjoint. The postcondition is a predicate on the input memory h_0, the return value (unit value () in this case), and output memory h_1. It states that the function only modifies dst, thus leaving inp (or any other array that is disjoint from dst) unchanged, and that the contents of dst in h_1 match the specification function applied to the contents of inp. Thus, no matter what algorithmic or low-level optimizations sha256_impl implements, once F⋆ verifies it with the above signature, its output is guaranteed to be consistent with the specification (as_seq is a Low⋆ library function that returns the contents of an array in a memory as a functional sequence).

Side-channel resistance. Following the methodology prescribed in Low⋆ [48], HACL⋆ algorithms are implemented with secrets modeled as abstract, constant-time integers. Indeed the type u_8 in the code listing for SHA256 spec above is the secret byte type. Thus, if the program type checks, it is guaranteed that the algorithm implementations cannot branch on secrets or use them as array indices, thus preventing the timing and memory access based side-channel leaks. In the ghost code (specification and proofs), the contents of the secret bytes may be inspected via coercions. We refer the reader to [65] for more HACL⋆ details.

4 DICE⋆ Engine

In this section, we present the DICE engine implementation in DICE⋆.

Verified properties. We prove that the CDI computation is functionally correct (as per Eq. 1). We also prove that the implementation does not leak secrets through heap by proving that: (a) it is memory-safe, (b) it does not allocate any memory on the heap, and (c) it only modifies CDI. Disallowing heap allocations guarantees that there are no memory leaks and secret leakage through dynamically-allocated memory.

```
val t : Type
val t_rel : Preorder.preorder (seq (erased t))
type state = {
  ghost_state : pointer (erased t) t_rel;
  cdi : array sbyte 32ul;
  l0_binary_size : u32;
  l0_binary : b:array sbyte l0_binary_size{
    eternal ghost_state ∧ eternal cdi ∧ eternal b ∧
    disjoint [ghost_state; cdi; l0_binary]
}}
val get_st () : state
val uds_len : i: u32 {0ul < i ∧ hashable i}
val uds_bytes : erased (lseq sbyte uds_len)
val uds_enabled (h:mem) : prop
val stack_cleared (h:mem) : prop
val read_uds (out:array sbyte uds_len) : Stack unit
  (requires λh → uds_enabled h ∧ live h out ∧ stack_array out)
  (ensures λh0 _ h1 →
    modifies out h0 h1 ∧ as_seq h1 out == uds_bytes)
val disable_uds () : Stack unit
  (requires λh → uds_enabled h)
  (ensures λh0 _ h1 →
    (¬ uds_enabled h1) ∧ modifies (get_st ()).ghost_state h0 h1)
val clear_stack () : Stack unit
  (requires λh → ¬uds_enabled h)
  (ensures λh0 _ h1 →
    (¬ uds_enabled h1) ∧ stack_cleared h1 ∧
    heap_arrays_except_ghost_state_are_preserved h0 h1)
```

Figure 3: Platform-agnostic interface used by DICE engine

Since Low⋆ only models a well-behaved subset of C, it does not allow us to reason about the (absence of) secret leaks via deallocated stack frames. Instead, we model an abstract clear_stack function, which is implemented natively in a platform-specific manner, and call this function to clear the stack memory just before transferring control to L0. Since it is not connected to the Low⋆ memory-model, it has to be manually audited to ensure that it is the last call in the DICE engine implementation. Finally, we also prove that the implementation is side-channel resistant.

Some aspects of the DICE engine are platform specific; for example, accessing and disabling UDS, primitives for erasing memory, and even the location of the CDI in the memory. To make the DICE engine implementation general and portable, we design a platform-agnostic interface (Section 4.1) against which we implement the core DICE engine (Section 4.2). While we provide a model F⋆ implementation of the interface, the extracted DICE engine C code is linked with a native, platform-specific implementation of it. This native implementation is part of our TCB.

4.1 Platform-Agnostic Interface

Figure 3 shows the platform-agnostic interface used by the DICE engine. The interface defines a state record type that exports the CDI array (a secret bytes array of length 32) and the L0 binary to the DICE engine. For driving the specifications about disabling UDS and clearing the stack, the state type also contains a pointer (i.e. an array of length 1) to an erased t, where t is an abstract type in the interface; the erased type constructor ensures that the type is safely erased during extraction. The interface associates a preorder t_rel with the ghost state pointer; F*'s theory of monotonicity [12] enforces that the contents of the pointer evolve as per t_rel. The refinement formula on the l0_binary field captures the invariant that all arrays in state are (a) pairwise disjoint, and (b) *eternal*, i.e. they are allocated on the heap and are never freed. The get_st API provides a way to get the state.

The interface exports the abstract uds_enabled and stack_cleared predicates—as we will see later, the DICE engine specification includes both of these in its postcondition. As we remarked earlier, the stack_cleared predicate is not connected to the memory model. The interface provides three main functions:

- read_uds provides access to the UDS; it copies the UDS into the argument array out. Its precondition requires the callers to prove that (i) access to UDS is enabled and (ii) out is a stack-allocated array that is live in the input memory. The postcondition of read_uds ensures that (a) it does no heap allocations (the Stack effect), (b) it only modifies out, and (c) the contents of out in the final memory are same as the (ghost) UDS bytes.

- disable_uds disables access to the UDS. Its postcondition ensures that it only modifies the ghost state, preserving contents of all other arrays.

- clear_stack clears the stack memory region in a platform-specific way. Its precondition requires that the UDS access is disabled. Its postcondition ensures that the stack_cleared predicate holds, and all the heap arrays, except the ghost state, are preserved in the final memory. Because ghost state is erased during extraction to C, clear_stack preserves all heap arrays, such as CDI.

The predicate heap_arrays_except_ghost_state_are_preserved is defined as:

```
let heap_arrays_except_ghost_state_are_preserved (h0 h1:mem) =
  let s = get_st () in
  ∀a len (b:array a len).
    (heap_array b ∧ disjoint [b; s.ghost_state] ∧ live h0 b) ⟹
    (as_seq h0 b == as_seq h1 b ∧ live h1 b)
```

Through abstraction, the interface enforces several properties in the DICE engine that uses it. First, access to UDS cannot be enabled after it is disabled. Indeed, only when the device reboots, will the access to UDS be enabled again. Second, heap or stack arrays cannot be modified by the DICE engine after clear_stack is called. stack_cleared is an abstract predicate, and the clear_stack function provides it as a postcondition on its output memory. The interface provides no other functions or lemmas for stack_cleared. Thus, if the DICE engine modifies memory in any way after clear_stack is called, it will not be able to prove stack_cleared in the final memory before returning. Third, clear_stack enforces that access to UDS must be disabled before its invocation. As a result, the interface enforces the following coding discipline on the DICE engine: it should read the UDS in a stack-allocated buffer, compute CDI, disable access to UDS, clear the stack, and return.

Model implementation of the interface. Figure 4 shows the model implementation of the platform-agnostic interface in F*. The implementation defines type t to be a pair of two booleans, the first indicates whether access to UDS is enabled, and the second indicates whether the stack has been cleared. The type t_rel enforces the aforementioned coding discipline: if access to UDS is enabled, then it may be disabled (the first transition from (true, _) to (false, _)); if access to UDS is disabled, then the stack may be cleared (the second transition), and the ghost state remains unchanged for all other transitions. The implementation defines a module-level variable of type state that is returned by the get_st function.

4.2 DICE Engine Implementation

We prove the following top-level specification for the DICE engine in DICE*:

```
let cdi_spec (h:mem) =
  let st = get_st () in
  as_seq h st.cdi == (∗ Functional spec for the CDI contents ∗)
    Spec.HMAC.hmac SHA2_256
      (Spec.Hash.hash SHA2_256 uds_bytes)
      (Spec.Hash.hash SHA2_256 (as_seq h st.l0_binary))

val dice_main () : Stack unit (requires λh → uds_enabled h)
(ensures λh0 () h1 →
  cdi_spec h1 ∧ (¬ uds_is_enabled h1) ∧ stack_cleared h1 ∧
  heap_arrays_except_cdi_and_ghost_state_are_preserved h0 h1)
```

The predicate cdi_spec specifies that the contents of the CDI buffer satisfies Eq. 1 using specifications about cryptographic primitives from HACL*. The dice_main function is in the Stack effect and requires that access to UDS is enabled when it is called. Its postcondition ensures that in the final memory CDI satisfies cdi_spec, access to the UDS is disabled, and stack_cleared is true. It also ensures that contents of all other heap arrays, except ghost state, are preserved. (Note that ghost state is erased at extraction). Thus, our DICE engine implementation is functionally correct, and does not leak secrets through memory or other interfaces such as network, disk,

```
type t = bool & bool
let t_rel = λs1 s2 → length s1 == length s2 ∧ (length s1 > 0 ⟹
   (let t1 = reveal (index s1 0) in
    let t2 = reveal (index s2 0) in
    match t1, t2 with
    | (true, _), (false, _)
    | (false, _), (false, true) → ⊤
    | _ → t1 == t2
let state_var : state = ... (∗ allocate the arrays ∗)
let uds_enabled h = fst (get h state_var.ghost_state)
let stack_cleared h = snd (get h state_var.ghost_state)
...
```

Figure 4: F⋆ implementation of platform-agnostic interface

```
let compute_cdi () =
  push_frame (); (∗ Low⋆ construct for stack frame creation ∗)
  let uds = alloca 0x00 uds_len in
  read_uds uds;
  let uds_hash = alloca 0x00 32ul in
  let l0_hash = alloca 0x00 32ul in
  Hacl.Hash.SHA2.hash_256 uds uds_len uds_hash;
  Hacl.Hash.SHA2.hash_256 st.l0_binary st.l0_binary_size l0_hash;
  Hacl.HMAC.compute_sha2_256 st.cdi uds_hash 32ul l0_hash 32ul;
  pop_frame ()

let dice_main () = compute_cdi (); disable_uds (); clear_stack ()
```

Figure 5: F⋆ implementation of DICE main function

etc. (because Stack effect does not permit any I/O). Figure 5 shows the implementation of dice_main.

5 DICE⋆ L0

This section presents the L0 implementation in DICE⋆.

Verified properties. Besides memory safety, we prove functional correctness for the outputs (DeviceID$_{pub}$ and its CSR, AliasKey pair, and AliasKey$_{pub}$ certificate). Functional correctness ensures that the code does not inadvertently leak secrets (CDI or DeviceID$_{priv}$) into these arrays. We also prove that our implementation does not leak secrets through the heap: it is memory safe, does not perform any heap allocations, and only modifies the necessary output arrays (as per their functional specifications). Secrets are modeled using the (abstract) type sbyte (as described in Section 3.3), which ensures that the code is side-channel resistant. Because our serializers (Section 5.2) are written over public bytes, F⋆ type-safety ensures that the L0 implementation does not serialize any secrets.

X.509 certificates introduce a new attack surface in L0. For instance, implementing the complex ASN.1 encoding format used by X.509 directly in C leaves open the possibilities of low-level exploitable memory errors. Furthermore, an *insecure* X.509 serializer (as defined in Section 5.1) could allow an attacker to break measured boot. For example, if the X.509 implementation is not injective, then an attacker could download a malicious L1 image on the device and exploit this non-injectivity to generate a certificate with the FWID of a valid L1 image.

General purpose X.509 libraries are large and complex, and come without any formal guarantees of correctness and security. Unsurprisingly, these are often the source of high-profile security vulnerabilities [1,4,5]. At the same time, L0 functionality requires only a subset of X.509 features (datatypes, extensions and cryptographic identifiers). To avoid the complexity of full X.509, the DICE specification recommends using a custom DICE-specific X.509 library (Section 7.3 in [58]). Therefore, we have built a formally verified, secure X.509 certificate serialization library that contains all features necessary for implementing L0. Extending this library to support more datatypes and encodings is an interesting future work.

Our X.509 library is built using LowParse [49], a library of parser combinators written in F⋆. Section 5.1 presents an overview of LowParse, Section 5.2 describes our extension to LowParse to support (a fragment of) X.509, and Section 5.3 presents a formally verified L0 implementation.

5.1 LowParse Overview

LowParse defines combinators for parsers and serializers capturing their correctness and security properties in the types. Given a set of valid messages V, the library defines a notion of *secure parsers* as parsers that are *complete*, i.e. accepting at least one binary representation of each message, and *non-malleable*, i.e. accepting at most one binary representation of each message. A *secure serializer* is the mathematical inverse of a secure parser (considering the parser to be a function from bytes to $V \cup \{\perp\}$, where $\perp$ denotes the error value).

By building our X.509 library using LowParse, we formally verify that our parsers and serializers are also secure. This means, for example, that our X.509 serializations are injective and, hence, the kind of L1 image impersonation attacks outlined above are not possible.

The LowParse architecture consists of a specification layer, where parser and serializer specifications are written in the pure fragment of F⋆ (using functional sequences and mathematical integers), and a low-level implementation layer written in Low⋆. The security proofs are done on the specification layer, while the low-level implementations are proven memory safe and functionally correct w.r.t. the specifications. During extraction, the specifications and the proofs are erased, and the low-level implementations are extracted to C.

The F⋆ type for parser specification is:

```
type pbyte = pu8 (∗ the type of public bytes ∗)
type parser (t:Type) (k:meta) =
  p:(input:seq pbyte → Ghost (option (t ∗ l:nat{l ≤ length input})))
    { parser_prop k p }
```

The parser specification parser t k is a ghost function that takes as input a sequence of bytes, and either returns an error (the value None), or a tuple with a value of type t and the number of consumed bytes. The refinement parser_prop k p ensures that the parser specification p satisfies properties specified by the metadata k, such as the non-malleability property.

The F* type for serializer specification is:

```
type serializer #t #k (p:parser t k) =
  s:(t → Ghost (seq pbyte)){∀ x. p (s x) == Some (x, length (s x))}
```

The serializer specification serializer p, indexed by the corresponding parser specification p, is a ghost function that serializes a value x of type t into a sequence of bytes such that parsing these bytes using p returns the same value v and consumes all the bytes in the sequence.

The Low* type of a low-level serializer implementation is:

```
type serializer32 #t #k (#p:parser t k) (s:serializer p) =
  x:t → b:array pbyte → pos:u32 → Stack u32
  (requires λh →
    live h b ∧ v pos + Seq.length (serialize s x) ≤ length b)
  (ensures λh0 len h1 → modifies b h0 h1 ∧
    as_seq h1 b ==
    replace (as_seq h0 b) (v pos) (v (pos + len)) (serialize s x))
```

The low-level serializer implementation serializer$_{32}$ s takes as input a value x of type t, an array of bytes b, and a position pos in b at which to serialize x, and returns the number of serialized bytes. The precondition requires that the array b is live and is large enough to store the serialization of x. Note that the specification function v is used to coerce a u$_{32}$ to a mathematical integer. The postcondition ensures that only the input array b is modified, no heap allocations are performed (specified via the Stack effect on the return type), and the len bytes of b starting at pos equal the serialization of x as specified by the serializer specification s, which ensures the functional correctness of the low-level serializer implementation.

Based on these types, LowParse defines *combinators*, which are higher-order functions that compose basic parsers and serializers into parsers and serializers for composite types. For example, the serialize_nondep_then combinator takes as input two serializer specifications s$_1$ and s$_2$ for types t$_1$ and t$_2$, resp., and builds a serialize specification for t$_1$ $*$ t$_2$ by invoking s$_1$ followed by s$_2$.

Ramananandro et al. [49] also present the EverParse framework that uses the LowParse combinators to auto-generate parsers and serializers from message formats specified in a domain-specific language. The paper also describes a functional-implementation layer (in addition to the specification layer and the low-level layer discussed above). We do not use these features, and refer the reader to [49] for more details. Instead, we focus on parser and serializer specifications, and low-level serializer implementation for X.509.

5.2 X.509 Serialization

The X.509 standard [24] describes the structure for public key certificates. An X.509 certificate contains basic fields such as a serial number, version, signature algorithm and value, and public key info, as well as optional extensions. The certificate structure is expressed in the Abstract Syntax Notation One (ASN.1) language. ASN.1 defines datatypes, such as integer, boolean, sequence, bitstring, octet string, and syntax for describing message formats using their composition. It also defines several binary encoding rules, such as Distinguished Encoding Rules (DER) [30], which is used by X.509 certificates.

DER encodes every message, including the basic types, in a Tag-Length-Value (TLV) format. The value bytes encode the message, which could be a primitive ASN.1 value or another TLV triplet. The length bytes encode the length of the value bytes, and tag is a one-byte value encoding the type. Both the value and length DER encodings are variable length.

Extending LowParse with backward serializers. LowParse supports serializing variable-length data using finalizers. A finalizer takes as input an array, with the precondition that the array contains a placeholder for the length of the data followed by the serialization of the data itself. The finalizer computes and writes the length in the placeholder, providing an appropriate postcondition.

However, finalizers are not suitable for DER as they require placeholders for serialization of the lengths. In the DER TLV format, the size of the length encoding depends on the value of length itself. Hence, determining the size of the placeholders for lengths requires making a pass over the message to compute its length before serializing the message itself. Since every DER encoding is TLV, this means making multiple passes on the sub-messages in a naïve implementation. One could optimize this to one pass by computing a length structure isomorphic to the message, but this requires changing the serializer type in LowParse to pass this additional argument—a suboptimal choice for fixed-length formats and a pervasive change to the LowParse library.

Thus, to support variable-length data in ASN.1 DER, we extend LowParse with low-level *backward serializers*. Backward serializers provide an elegant solution to the problem. Instead of serializing messages forward from the beginning of the array, backward serializers serialize messages backward from the end of the array. They return the number of bytes serialized, which can then be serialized by the caller at the beginning of the serialized message. This allows us to build generic TLV serializers, without explicitly requiring length computations. The type of the backward serializers is:

```
type serializer32_backwards #t #k (#p:parser t k) (s:serializer p) =
  x:t → b:array pbyte → pos:u32 → Stack u32 (requires λh →
    live h b ∧ Seq.length (serialize s x) ≤ v pos ≤ length b)
  (ensures λh0 len h1 → modifies b h0 h1 ∧ as_seq h1 b ==
    replace (as_seq h0 b) (v (pos − len)) (v len) (serialize s x))
```

The pos argument to the backward serializers is the ending position in the array. The postcondition establishes that the contents of the array between [pos − len, pos) are the serialized bytes (functionally correct w.r.t. the forward specification serializer s). Since we do not change the parser and serializer specifications, our low-level backward serializers enjoy the same security properties as before.

Using backward serializers, we implement a generic TLV serializer as follows:

```
let serializer32_tlv_backwards s32 x tag b pos =
  let l_value = s32 x b pos in (∗ serialize value ∗)
  let l_length = (∗ serialize length ∗)
    serialize32_len_backwards l_value b (pos − l_value) in
  let l_tag = (∗ serialize tag ∗)
    serialize32_tag_backwards tag b (pos − l_value − l_length) in
  l_value + l_length + l_tag (∗ return number of bytes written ∗)
```

We also extend LowParse with combinators for backward serialization. For example, the serialize$_{32}$_nondep_then_backwards combinator takes as input two backward serializers s$_1$ and s$_2$, and invokes s$_2$ followed by s$_1$.

ASN.1 serializers. We program parser and serializer specifications, and low-level backward serializers for the DER encoding of the ASN.1 fragment needed to implement L0. The parser and serializer specifications are proven secure, and the low-level serializers are proven memory safe and functionally correct w.r.t. the specification, all in the (extended) LowParse framework.

Our implementation supports ASN.1 lengths in the range $[0, 2^{32})$, and all ASN.1 tags. For ASN.1 primitive types, it supports: BOOLEAN, NULL, non-negative INTEGER, OCTET_STRING, PRINTABLE_STRING, IA5_STRING, BIT_STRING, OBJECT_IDENTIFIER (OID), and specific values for GENERALIZED_TIME and UTC_TIME (used in the X.509 validity field). For structured types, it supports SEQUENCE, as well as empty and singleton SET. We support implicit and explicit tagging over both primitive and structured types.

For the supported ASN.1 primitive types, we first define their Low* representations:

```
let datatype_of_asn1_type (a:asn1_type) = match a with
| BOOLEAN → bool
| OCTET_STRING → (len: u32 {len < 2^32 − 6} ∗ array pbyte len)
| BIT_STRING → bit_string_t
| OID → oid_t
... (∗ definition for other ASN.1 primitive types ∗)
```

The Low* representation for OCTET_STRING is a dependent pair of a length len$\in[0, 2^{32}-6)$ and an array of (public) bytes of length len. The invariant on len ensures that the length of the corresponding TLV message is less than 2^{32}. Invariants on other types are more involved; e.g., the Low* representation for BIT_STRING may contain unused bits that must be zero.

Then, we define the parser and serializer specifications, and the low-level serializer implementation on these representation types to match their DER encoding. For example, the ASN.1 BOOLEAN values TRUE and FALSE are serialized as 8-bit unsigned integers 0xFFuy and 0x00uy respectively. The parser specification for the BOOLEAN type is:

```
val parse_asn1_boolean
  : parser (datatype_of_asn1_type BOOLEAN) boolean_meta
let parse_asn1_boolean = parse_u8
  `parse_filter` (λ b → b = 0xFFuy || b = 0x00uy) `parse_synth`
  (λ b → match b with | 0xFFuy → true | 0x00uy → false)
```

Here, a `f` b is infix notation for f a b, and parse_u$_8$, parse_filter, and parse_synth are LowParse combinators [49]. For each type, we also build its TLV serializer using the generic TLV serializer sketched above.

X.509 serializers. We follow a similar methodology for X.509 structures needed for L0 [58]. For example, the X.509 standard [24] defines AlgorithmIdentifier and SubjectPublicKeyInfo as:

```
AlgorithmIdentifier ::= SEQUENCE {
  algorithm OBJECT IDENTIFIER,
  parameters ANY DEFINED BY algorithm OPTIONAL }

SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  subjectPublicKey BIT STRING }
```

The parameters field of AlgorithmIdentifier is algorithm-specific. For Ed25519 [32], for example, the algorithm should be the OID id−Ed25519 and the parameters field should be absent. The subjectPublicKey field in SubjectPublicKeyInfo for Ed25519 must be a 33-byte bit string where the leading byte is set to zero; this leading byte specifies the unused bits in the last byte. We define the corresponding types and serializer specifications as follows:

```
type the_oid oid = o:oid{o == oid} (∗ Singleton OID type ∗)
type alg_identifier_payload_t = the_oid OID_ED25519
let serialize_alg_identifier_payload = (∗ Spec serializer ∗)
  serialize_the_oid OID_ED25519
let serialize_alg_identifier = (∗ SEQUENCE tagging ∗)
  serialize_envelope_sequence serialize_alg_identifier_payload

type bit_string_with_len_and_unused l n =
  b:bit_string_t{length b == l ∧ unused b == n}
type subject_public_key_info_payload_t = {
  algorithm : envelope SEQUENCE alg_identifier_payload_t;
  subject_public_key : bit_string_with_len_and_unused 33ul 0ul }
let serialize_subject_public_key_info_payload =
  serialize_alg_identifier `serialize_nondep_then`
  serialize_bit_string_with_len_and_unused 33ul 0ul
let serialize_subject_public_key_info = serialize_envelope_sequence
  serialize_subject_public_key_info_payload
```

Following this scheme, we define parsers and serializers of all the X.509 structures required for DeviceID CSR and AliasKey certificate in accordance with the DICE certificate standard [58].

X.509 extension for L0. The DICE certificate standard [58] defines an X.509 extension for L0. The extension describes how the device identity, consisting of the DeviceID$_{pub}$ and FWID, should be embedded inside the AliasKey certificate.

```
TCG−DICE−FWID ::== SEQUENCE {
  TCG−DICE−fwid OBJECT IDENTIFIER,
  SEQUENCE CompositeDeviceID }
CompositeDeviceID ::== SEQUENCE {
  version INTEGER,
  SEQUENCE SubjectPublicKeyInfo,
  SEQUENCE FWID }
FWID ::== SEQUENCE {
  hashAlg OBJECT IDENTIFIER,
  fwid OCTET STRING }
```

Similar to SubjectPublicKeyInfo, we define parser and serializer specification as well as low-level serializer implementation for this extension.

5.3 L0 Implementation

The F* type for the core L0 function is shown in Figure 6. The function l0_core takes as input (a) CDI and FWID, (b) HKDF labels to be used in the derivation of the DeviceID key pair and AliasKey pair, (c) the DeviceID CSR and AliasKey certificate details (such as the version, serial number, etc.), and (d) arrays for writing DeviceID$_{pub}$, AliasKey$_{pub}$, AliasKey$_{priv}$, DeviceID CSR, and AliasKey certificate. Because the DICE specification does not specify what exactly constitutes a FWID measurement, we take FWID as an input. The implementation can easily be adapted to support specific measurement functions computed inside l0_core.

The precondition requires that all the arrays are live and pairwise disjoint, and that the length of the CSR and certificate arrays equals the size of the (serialized) certificate and CSR, respectively; we provide auxiliary functions to compute the exact size for the certificate and CSR. Currently, we enforce the length requirement as a precondition, but other implementations, such as runtime checks, are possible.

The function has the Stack effect, ensuring that it does not perform any heap allocations. Its postcondition ensures that the function only modifies the contents of the DeviceID$_{pub}$, AliasKey pair, CSR, and certificate arrays, in accordance with their functional specifications. Below we show the functional specification for AliasKey certificate, which specifies the contents of the ak_crt_arr in terms of the specification-level serializer for the AliasKey certificate. The specification functions for key derivation (e.g. dk_spec below) integrate with the key derivation specifications from HACL*.

```
let ak_crt_post cdi fwid dk_label ak_label ak_crt ak_crt_arr h0 h1 =
  let dk_pub, dk_priv = dk_spec cdi fwid dk_label h0 in
  let ak_crt =
    ak_crt_spec cdi fwid dk_pub dk_priv ak_label ak_crt h0 in
  (∗ Functional correctness for the AliasKey certificate array ∗)
  as_seq h1 ak_crt_arr == serializer_ak_crt `serialize` ak_crt
```

The implementation of l0_core derives the DeviceID and AliasKey using the HKDF and Ed25519 libraries from HACL*. The implementation then creates a DeviceID$_{pub}$ CSR Low* value signed using the DeviceID$_{priv}$ and serializes it into the dk_csr_arr using its low-level serializer. Finally, it creates the AliasKey certificate value, signed using the DeviceID$_{priv}$, and serializes it in ak_crt_arr. In all these cases, it is proved that the serialized bytes match their functional specifications.

```
val l0_core (cdi:array sbyte 32) (fwid:array pbyte 32)
  (dk_label_len:u32) (dk_label:array pbyte (v dk_label_len))
  (ak_label_len:u32) (ak_label:array pbyte (v ak_label_len))
  (dk_csr:csr_t) (ak_crt:crt_t) (dk_pub: array pbyte 32)
  (ak_pub:array pbyte 32) (ak_priv:array sbyte 32)
  (dk_csr_len:u32) (dk_csr_arr:array pbyte (v csr_len))
  (ak_crt_len:u32) (ak_crt_arr:array pbyte (v crt_len)) : Stack unit
(requires λh → ... ∧ (∗ liveness and disjointness of arrays ∗)
  (∗ label lengths are valid HKDF lengths ∗)
  is_hkdf_label dk_label_len ∧ is_hkdf_label ak_label_len ∧
  (∗ the CSR and certificate arrays have the required lengths ∗)
  dk_csr_pre dk_csr dk_csr_len ∧ ak_crt_pre ak_crt ak_crt_len)
(ensures λh0 () h1 → modifies
  [dk_pub; ak_pub; ak_priv; dk_csr_arr; ak_crt_arr] h0 h1 ∧
  (∗ Functional spec for the DeviceID public key ∗)
  dk_post cdi dk_label dk_pub h0 h1 ∧
  (∗ Functional spec for the AliasKey pair ∗)
  ak_post cdi fwid ak_label ak_pub ak_priv h0 h1 ∧
  (∗ Functional spec for the DeviceID CSR ∗)
  dk_csr_post cdi dk_label dk_csr dk_csr_arr h0 h1 ∧
  (∗ Functional spec for the AliasKey certificate ∗)
  ak_crt_post cdi fwid dk_label ak_label ak_crt ak_crt_arr h0 h1)
```

Figure 6: Signature of the core L0 function. Identifiers with prefix ak_ and dk_ refer to AliasKey and DeviceID resp.

Declassification of public data. The low-level implementations in the HACL* library operate exclusively on secret bytes; e.g., the public key pair derivation function returns even the public key in an array of secret bytes. Because secret and public bytes are different types, type-safety in F* does not allow copying public keys in secret byte arrays directly into the (public) output arrays. Thus, we need to explicitly declassify three public keys and two signatures: DeviceID$_{pub}$, AliasKey$_{pub}$, the digest of the DeviceID$_{pub}$ as the authority key identifier used in the AliasKey certificate extension, DeviceID$_{pub}$ CSR signature, and AliasKey certificate signature. We model declassification using a trusted function as follows:

```
let declassify_spec len (s:lseq sbytes len) : lseq pbytes len = ...
val declassify (len:u32) (src:array sbyte len) (dst:array pbyte len)
  : Stack unit
    (requires λh → live h src ∧ live h dst ∧ disjoint [src; dst])
    (ensures λh0 () h1 → modifies dst h0 h1 ∧
      as_seq h1 dst == declassify_spec (as_seq h0 src))
```

The extracted DICE* L0 code is linked against a native implementation of the declassification function, which can use either memcpy or a verified memcpy extracted from Low*.

In general, such declassifications need to be manually audited to ensure that only the intended data is declassified. However, in our case, precise functional specification of all the output arrays ensures that the verification will fail if incorrect data is declassified. Since our code does not use the heap and explicitly clears the stack, all outputs are via argument arrays whose contents are precisely specified in the postconditions. For example, the dk_post specification used in l0_core's postcondition (in Figure 6) explicitly states that the contents of the array dk_pub are same as declassifying the output of the function derive_dk_pub_spec:

```
let dk_post (cdi:array sbyte 32) (dk_label: array pbyte)
  (dk_pub:array pbyte 32) (h0 h1:mem) =
  as_seq h1 dk_pub
    == declassify_spec 32 (derive_dk_pub_spec cdi dk_label)
```

Thus, a bug in declassification, e.g. declassifying the private key instead of public key, would result in a verification failure for this postcondition.

6 DICE* Implementation

Table 1 shows the lines of code (LOC) for DICE*. The DICE engine implementation in DICE* consists of 533 lines of (commented) F* code, including the specifications, implementations, and proofs, which extract to 205 lines of C code. DICE* L0 implementation consists of 24,241 lines of F* code, 16,564 of those implementing the ASN.1/X.509 library. The L0 implementation extracts to 5,051 lines of C.

Table 1 also shows the verification times for DICE*. The measurements are taken on an HP Z840 workstation with Intel® Xeon® CPU E5-2699 v4 (2.20GHz) and 64GB RAM. The time measurements are with parallelism provided by modular verification, verifying L0 sequentially takes 26m2s. Note that the LOC and verification times in Table 1 do not include HACL* and LowParse.

While the DICE engine implementation was relatively straightforward to verify, to scale the verification to ASN.1/X.509 library and the L0 implementation, we used the following proof-engineering mechanisms:

Abstraction via F* interfaces. We use F*'s interface mechanism to abstract away irrelevant definitions from the SMT solver's proof context, thereby reducing the size of the SMT queries. For example, we declare the type of the definition parse_asn1_boolean (Section 5.2) in the ASN.1/X.509 library in an interface file as follows:

```
val parse_asn1_boolean
    : parser (datatype_of_asn1_type BOOLEAN) boolean_meta
```

For the clients, the definition of parse_asn1_boolean is not important—it is sufficient that the low-level boolean serializer implementation provides this spec in its type. Therefore, we add the implementation of parse_asn1_boolean in the separate implementation file. When F* verifies its clients, only the interface file is in scope, and hence, the implementation details are hidden from the client proofs.

Table 1: LOC and verification time for DICE*

	F* LOC	C LOC	Verification Time
DICE Engine	533	205	1m10s
L0	24,241	5,051	11m9s

Proof decompositon. When verifying a function like l0_core (Figure 6), F* and the Z3 SMT solver need to reason about multiple proof aspects, including arrays, secret bytes, cryptography, and serialization. When all of these proof obligations are sent as a single query to the SMT solver, the proofs sometimes don't scale. We get around this by decomposing functions with large proof obligations into auxiliary lemmas with smaller proof obligation. For example, in the case of l0_core, we prove the modifies theory related properties in a separate lemma lemma_l0_core_modifies:

```
let lemma_l0_core_modifies (pub_t: Type) (sec_t: Type)
    (ak_pub:array pub_t 32) (ak_priv:array sec_t 32) (h0 h1:mem)
    ... (∗ other buffers and intermediate memory states ∗) ...
: Lemma ((∗ mod. spec. between intermediate memory states ∗) ...
∧ modifies [ak_pub; ak_priv; ...] h0 h1) = ()
```

Separating out proof obligations in this manner significantly decreases the total verification time of l0_core.

Using meta-programming to discharge proof obligations. F* also has a meta-programming and tactics framework [39] using which programmers can write F* programs to inspect and prove properties of other F* programs. The metaprograms are evaluated by the F* typechecker at the time of typechecking. For proofs that involve large computations, we used metaprogramming to carry out those computations and simplify the proof obligations before they are sent to the SMT solver. This provided significant speedups in some cases.

7 Evaluation

In this section, we evaluate DICE* by comparing it against an unverified, hand-written DICE implementation in terms of boot time and binary size. The goal of the evaluation is to ensure that there are no unforeseen overheads of using verified code. We evaluate DICE* on the STM32H753ZI microcontroller unit (MCU) from ST Microelectronics [11]. The STM32H753ZI micro-controller is based on the ARM Cortex-M7 family of CPUs. It operates at 480 Mhz; it has high-speed embedded memories, including 2MB of dual bank flash and 1MB of RAM, and various other interfaces and peripherals.

Section 7.1 describes the bootloader and the platform-specific interface of DICE* for STM32H753ZI, and Section 7.2 compares DICE* against an unverified, hand-written DICE implementation in terms of binary size and boot time on STM32H753ZI.

7.1 DICE* for STM32H753ZI

We implement the bootloader and the platform-specific interface of DICE* for STM32H753ZI using a hardware security feature called *secure access mode*. This mode enables the development of security-critical services such as bootloaders that execute in isolation just after reset. Specifically, during manufacturing, a region in flash memory can be configured as a *secure area*, and can be provisioned with code and data of a secure service. The hardware guarantees that this area can only be accessed while the CPU is in secure access mode, which the CPU enters just after reset. While the CPU is in secure mode, the CPU ignores all debugging events. Once the CPU exits this mode (using a special instruction), reads to this area return zero, writes are ignored, and any attempt to execute code from this area generate errors. The secure area is also erase protected; i.e., no erase operations on a sector in this area are permitted.

We implement the bootloader using secure access mode as follows. We store the bootloader, DICE-engine image, which includes unverified platform-specific interface, and the public key used by the DICE engine in a secure area in flash memory. The bootloader receives control after a reset. It checks if UDS has already been provisioned at a pre-defined location in the secure area. If the UDS has not been provisioned, then the bootloader generates a fresh UDS by sampling a hardware RNG, and stores the UDS in the secure area.

Next, the bootloader transfers control to the DICE engine. The DICE engine, as per specification, authenticates the L0 image, derives CDI and latches UDS by exiting the secure access mode. Finally, the control comes back to the bootloader which then transfers control to L0.

We implement the platform-specific interface of DICE* (Section 4.1) as follows:

- read_uds is implemented by copying UDS stored at a pre-defined address in secure area to a buffer in RAM.
- disable_uds is empty because there is no explicit mechanism to disable access to UDS on this MCU. Disabling access is the responsibility of the bootloader.
- clear_stack is implemented by erasing all registers (except the stack pointer), and erasing all regions in SRAM, which holds the stack.

The bootloader and the platform-specific interface of DICE* together contain 38 lines of assembly and 815 lines of C code. This code is part of our TCB.

7.2 Comparison with Unverified DICE

We compare the boot time and the binary size of DICE* with that of an unverified, handwritten DICE implementation. The hand-written implementation uses cryptographic primitives from mbedTLS [7], a cryptographic library commonly used in embedded systems. The two implementations match in all respects except elliptic curve p-256 [19] used for firmware authentication in DICE and generating certificates and CSRs in L0. While the hand-written implementation uses p-256, DICE* uses Ed25519. This is because mbedTLS currently does not support Ed25519, and HACL* does not currently support a side-channel free implementation of p-256.

Table 2: Boot time (milliseconds) for each layer and the binary size (KB) of unverified DICE (Unv. DICE) and DICE*

Layer	Boot time (ms)		Size (KB)	
	Unv. DICE	DICE*	Unv. DICE	DICE*
DICE engine	786	689	72	68
L0	313	208	92	92

Table 2 compares the boot time (measured in milliseconds) of two DICE layers in these implementations. In both layers, DICE* has better performance compared to the unverified implementation. In the DICE engine, the difference in boot times is due to the difference in the performance of P-256 and Ed25519 based image verification. This is consistent with previously reported performance of these curves [60]. All other operations in the DICE engine have comparable performance. In L0, the difference in boot time is due to the X.509 certificate serialization logic. Unverified code relies on X.509 support in mbedTLS, whereas verified code uses our X.509 custom library built using LowParse.

Table 2 also shows a comparison of the binary sizes. Binary size is an important metric, especially in embedded systems where the amount of flash memory is often limited. Both implementations have a comparable binary size.

In summary, DICE* compares favorably with the unverified implementation both on performance and binary size, and, thus, should form the basis for future DICE implementations.

8 Related Work

This paper presents a verified implementation of DICE [38, 55], which is an emerging industry standard for measured boot proposed by TCG. There are also efforts on developing attestation protocols based on DICE [29, 31] and extending DICE with new features to support secure firmware updates and re-provisioning of DICE-powered devices [62].

Hardware solutions for trusted computing such as TPM [28], ARM TrustZone [14] and Intel SGX [40] are not suitable for low-cost devices. Compared to the minimal hardware requirements of the DICE architecture, the hardware-based solutions designed for isolation and attestation of embedded devices, such as TyTAN [21], TrustLite [34], and Sancus [45,46], are complex and costly [37]. Software-based solutions for device attestation, such as SWATT [52], Pioneer [51], and VIPER [36], make impractical assumptions [15].

DICE* focuses on verification of memory-safety, full-functional correctness, and side-channel resistance for the DICE measured boot protocol. Cook et al. [23] use the CBMC model checker [35], extended with device-specific extensions, to prove memory-safety of the boot code used in the AWS data centers. Their boot code is not measured or authenticated boot, the stages in their code only locate, load, and launch the next stage. As a result, its guarantees, and the implementation complexity, are much weaker than DICE. Straznickas et al. [53], in what seems to be a work-in-progress, use the Coq theorem prover towards verifying functional-correctness and termination of a first-stage bootloader written in RISC-V assembly. Muduli et al. [44] use model checking to verify that (model of) a firmware loader only loads valid images. They cast the security property as a hyperproperty [22], modeling TOCTTOU attacks. Hristozov et al. [29] propose a runtime attestation scheme, augmenting DICE, to protect against (undetected) runtime compromise of the firmware code, an unlikely scenario with fully verified and memory-safe DICE*.

For X.509 certificate generation, we extended the LowParse framework [49], and provide memory-safe, functionally-correct, and secure ASN.1/X.509 serializers. Tullsen et al. [59] present verified encoders and decoders for a subset of ASN.1 required for vehicle-to-vehicle (V2V) messaging. However, they do not verify full-functional correctness, but only an approximation of it, called *self-consistency* which states that (a) a valid message that is encoded and decoded results in the same message, and (b) the decoder only accepts valid messages. They carry out the verification in the Software Analysis Workbench [26] tool. Ye et al. [63] focus on the Protocol Buffers data format and formally verify protobuf serializers and deserializers for functional correctness in Coq. Their work is based on Narcissus [25] that defines a non-deterministic data-format and derives verified encoders and decoders using a library of higher-order combinators, like in LowParse. The distinguishing feature of LowParse and our work is the security proof and the generation of C code from a verified implementation.

9 Conclusion

We have presented DICE*, an implementation of the DICE measured boot protocol that is provably memory-safe, functionally-correct, and side-channel resistant. A key component of DICE* is a secure X.509 library that generates DICE-compliant certificates and CSRs. We believe this implementation can form a more secure baseline for future implementations of the DICE architecture, avoiding bug-finding and fixing cycles. DICE* can be extended to further improve the security of measured boot e.g. by building verified implementations of hardware protection mechanisms underlying the DICE architecture, and of commonly used components in L0 firmware such as attestation and key exchange protocols.

References

[1] CVE-2014-0092. https://nvd.nist.gov/vuln/detail/CVE-2014-0092.

[2] CVE-2014-1568. https://nvd.nist.gov/vuln/detail/CVE-2014-1568.

[3] CVE-2016-0701. https://nvd.nist.gov/vuln/detail/CVE-2016-0701.

[4] CVE-2016-2108. https://nvd.nist.gov/vuln/detail/CVE-2016-2108.

[5] CVE-2020-0601. https://nvd.nist.gov/vuln/detail/CVE-2020-0701.

[6] CVE-2020-9434. https://nvd.nist.gov/vuln/detail/CVE-2020-9434.

[7] MbedTLS. https://tls.mbed.org.

[8] Micron CEC1702. https://www.microchip.com/wwwproducts/en/CEC1702.

[9] Microsoft Projet Cerberus. https://github.com/Azure/Project-Cerberus.

[10] NXP LPC5500. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc5500-cortex-m33:LPC5500_SERIES.

[11] STM32H753ZI. https://www.st.com/en/microcontrollers-microprocessors/stm32h743-753.html.

[12] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL), 2018.

[13] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1997.

[14] A Arm. Security technology-building a secure system using trustzone technology. *ARM Technical White Paper*, 2009.

[15] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[16] Elaine Barker et al. *Recommendation for key management: Part 1: General, SP 800-57 Part 1 Rev. 5*. National Institute of Standards and Technology, 2020.

[17] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[18] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[19] Simon Blake-Wilson, Nelson Bolyard, Vipul Gupta, Chris Hawk, and Bodo Möller. Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS). *RFC*, 4492, 2006.

[20] Hanno Böck. Wrong results with Poly1305 functions. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413, 2016.

[21] Franz Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: tiny trust anchor for tiny devices. In *52nd Design Automation Conference (DAC)*. ACM, 2015.

[22] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 2010.

[23] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In *30th International Conference on Computer Aided Verification (CAV)*, 2018.

[24] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and W. Timothy Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC*, 5280, 2008.

[25] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP):82:1–82:29, 2019.

[26] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing semantic models of programs with the software analysis workbench. In *Verified Software. Theories, Tools, and Experiments*, 2016.

[27] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In *19th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[28] Trusted Computing Group. TPM main specification level 2 version 1.2, revision 116. https://trustedcomputinggroup.org/resource/tpm-main-specification/, 2011.

[29] Stefan Hristozov, Johann Heyszl, Steffen Wagner, and Georg Sigl. Practical runtime attestation for tiny IoT devices. In *Proceedings of the 2018 Workshop on Decentralized IoT Security and Standards*, volume 18, 2018.

[30] ITU-T. X.690 information technology–ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER). Technical report, ITU, 2015.

[31] Lukas Jäger, Richard Petri, and Andreas Fuchs. Rolling DICE: Lightweight remote attestation for COTS IoT hardware. In *12th International Conference on Availability, Reliability and Security (ARES)*. ACM, 2017.

[32] Simon Josefsson and Jim Schaad. Algorithm identifiers for Ed25519, Ed448, X25519, and X448 for use in the internet X.509 public key infrastructure. *RFC*, 8410, 2018.

[33] Corey Kallenberg, Sam Cornwell, Xeno Kovah, and John Butterworth. Setup for failure: defeating secure boot. In *The Symposium on Security for Asia Network (SyScan)*, 2014.

[34] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *9th European Conference on Computer Systems (EuroSys)*. ACM, 2014.

[35] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014.

[36] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: verifying the integrity of peripherals' firmware. In *18th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2011.

[37] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Computers*, 67(3):361–374, 2018.

[38] Andrey Marochko, Dennis Mattoon, Paul England, Ronald Aigner, Rob Spiger (CELA), and Stefan Thom. Cyber-resilient platforms overview. Technical Report MSR-TR-2017-40, Microsoft, September 2017.

[39] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal

Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*. Springer, 2019.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[41] Microsoft. Secure the windows 10 boot process. https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process, 2018.

[42] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *29th USENIX Security Symposium*. USENIX Association, August 2020.

[43] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology (ICISC)*. Springer, 2006.

[44] Sujit Kumar Muduli, Pramod Subramanyan, and Sayak Ray. Verification of authenticated firmware loaders. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019.

[45] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22th USENIX Security Symposium*. USENIX Association, 2013.

[46] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3), 2017.

[47] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[48] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.*, 1(ICFP), 2017.

[49] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *28th USENIX Security Symposium*. USENIX Association, 2019.

[50] Steffen Schulz, André Schaller, Florian Kohnhäuser, and Stefan Katzenbeisser. Boot attestation: Secure remote reporting with off-the-shelf IoT sensors. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017.

[51] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *20th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2005.

[52] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. SWATT: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2004.

[53] Zygimantas Straznickas. *Towards a Verified First-Stage Bootloader in Coq*. Master thesis, Massachusetts Institute of Technology, 2020.

[54] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In *43rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2016.

[55] Trusted Computing Group. DICE. https://trustedcomputinggroup.org/work-groups/dice-architectures/.

[56] Trusted Computing Group. DICE Layering Architecture. https://trustedcomputinggroup.org/resource/dice-layering-architecture/.

[57] Trusted Computing Group. Hardware Requirements for a Device Identifier Composition Engine. Family 2.0, Level 00, Revision 78. https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf, March 22, 2018.

[58] Trusted Computing Group. Trusted Computing Group: Implicit Identity Based Device Attestation. Version 1.0, Revision 0.93. https://trustedcomputinggroup.org/resource/implicit-identity-based-device-attestation/, March 5, 2018.

[59] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. Formal verification of a vehicle-to-vehicle (V2V) messaging system. In *30th International Conference Computer Aided Verification (CAV)*. Springer, 2018.

[60] Roland van Rijswijk-Deij, Kaspar Hageman, Anna Sperotto, and Aiko Pras. The performance impact of elliptic curve cryptography on dnssec validation. *IEEE/ACM transactions on networking*, 25(2):738–750, 2016.

[61] Richard Wilkins and Brian Richardson. Uefi secure boot in modern computer security solutions. In *UEFI Forum*, 2013.

[62] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the internet of things. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[63] Qianchuan Ye and Benjamin Delaware. A verified protocol buffer compiler. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. ACM, 2019.

[64] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. Providing root of trust for ARM TrustZone using on-chip SRAM. In *4th International Workshop on Trustworthy Embedded Devices (TrustED)*. ACM, 2014.

[65] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.