An In-Depth Symbolic Security Analysis of the ACME Standard

Karthikeyan Bhargavan
karthikeyan.bhargavan@inria.fr
INRIA
Paris, France

Abhishek Bichhawat
abhishek.b@iitgn.ac.in
IIT Gandhinagar
Gandhinagar, Gujarat, India

Quoc Huy Do*
quoc-huy.do@sec.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Pedram Hosseyni
pedram.hosseyni@sec.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Ralf Küsters
ralf.kuesters@sec.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Guido Schmitz†
guido.schmitz@sec.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Tim Würtele
tim.wuertele@sec.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

ABSTRACT

The ACME certificate issuance and management protocol, standardized as IETF RFC 8555, is an essential element of the web public key infrastructure (PKI). It has been used by *Let's Encrypt* and other certification authorities to issue over a billion certificates, and a majority of HTTPS connections are now secured with certificates issued through ACME. Despite its importance, however, the security of ACME has not been studied at the same level of depth as other protocol standards like TLS 1.3 or OAuth. Prior formal analyses of ACME only considered the cryptographic core of early draft versions of ACME, ignoring many security-critical low-level details that play a major role in the 100 page RFC, such as recursive data structures, long-running sessions with asynchronous sub-protocols, and the issuance for certificates that cover multiple domains.

We present the first in-depth formal security analysis of the ACME standard. Our model of ACME is executable and comprehensive, with a level of detail that lets our ACME client interoperate with other ACME servers. We prove the security of this model using a recent symbolic protocol analysis framework called DY★, which in turn is based on the F★ programming language. Our analysis accounts for all prior attacks on ACME in the literature, including both cryptographic attacks and low-level attacks on stateful protocol execution. To analyze ACME, we extend DY★ with authenticated channels, key substitution attacks, and a concrete execution framework, which are of independent interest. Our security analysis of ACME totaling over 16,000 lines of code is one of the largest proof developments for a cryptographic protocol standard in the literature, and it serves to provide formal security assurances for a crucial component of web security.

CCS CONCEPTS

• **Security and privacy** → **Formal security models**; **Security protocols**; *Web protocol security*; *Digital signatures*; **Formal security models**; **Logic and verification**; **Security protocols**; **Web protocol security**; • **Networks** → **Protocol testing and verification**; **Protocol testing and verification**; **Formal specifications**; • **Theory of computation** → *Cryptographic protocols*; **Cryptographic protocols**.

KEYWORDS

formal protocol analysis and verification; public-key-infrastructure; certificate issuance

1 INTRODUCTION

The management of certificates for web servers used to be a very tedious and manual task. To relieve administrators from this burden, the *Internet Security Research Group (ISRG)* developed the *Automatic Certificate Management Environment (ACME)*, which provides a protocol to automate the process of domain ownership verification and certificate issuance. ACME was standardized by the *Internet Engineering Task Force (IETF)* as RFC 8555 [5] and is now supported by a wide range of certification authorities (CAs) (e.g., [21, 25, 33, 43, 57]) and many different web server tools (see [41] for a detailed list). In particular, *Let's Encrypt* [43], launched by the ISRG in 2015, was the first CA to implement ACME and has provided its service to all users for free (see also their CCS paper on the history of ACME [1]).

---

*Also with GLIWA GmbH.

†Also with Royal Holloway University of London.

By now, *Let's Encrypt* and ACME have become extremely popular and are widely used and trusted, with more than 1 billion certificates issued so far by *Let's Encrypt* alone [40], accounting for about 57% of certificates in active use on the web [1].

Since ACME is so widely used, a design flaw in ACME can have disastrous consequences. For example, a critical cryptographic flaw in an early draft of ACME [2] allowed an adversary to obtain certificates for domains not owned by the adversary (see Section 5 for details). This flaw was fixed in the published standard, but to ensure no such protocol flaws remain, it is important to formally verify the security of the ACME standard.

Formally Analyzing ACME. Two prior works analyzed early drafts of the ACME protocol using the symbolic protocol analyzers ProVerif and Tamarin [15, 36]. These analyses were able to automatically identify protocol weaknesses in early ACME drafts and verify their fixes. However, they only considered the core cryptographic mechanisms in the basic certificate issuance message flow for a single domain, resulting in high-level models of a few hundred lines that leave out many of the details of the 100-page ACME standard.

ACME relies on recursive control flows, unbounded data structures, and careful state management for long-running sessions that involve multiple asynchronous sub-protocols. For example, an ACME client can ask the ACME server for a certificate that covers a list of domains. The server has to iteratively go through this list and for each domain therein create a URL, which is an individual endpoint the client later has to connect to. For each such URL, client and server then asynchronously run a subprotocol consisting of several further message rounds in order to verify that the client owns the domain. In particular, the runs of these different subprotocols can interleave arbitrarily. When the ownership of all domains has been validated, the client can finally ask the server to issue a new certificate for the domain list and a given public key.

This complex set of interactions raises many open security questions that remained unanswered by prior formal analyses. For example, can an attacker participate in a series of asynchronous subprotocols with an ACME server and drive the server into a state where it would be willing to issue the attacker a certificate covering a domain that it does not own? Such questions have practical, real-world consequences. Recently, a severe flaw in the ACME server implementation *Boulder* (used, e.g., by *Let's Encrypt*) was discovered. This flaw was rooted in the incorrect processing of domain lists and allowed an attacker to obtain certificates for domains it does not own. As a consequence of this flaw, *Let's Encrypt* had to revoke more than 3 million certificates [35].

The goal of this work is to carry out an in-depth security analysis of the ACME standard that accounts for all these low-level protocol details, so that we can formally prove that an ACME implementation that faithfully follows the standard has strong security guarantees.

A Detailed Executable ACME Model in F*. Writing a detailed formal model for a large protocol standard like ACME is a challenging task. The protocol uses flexible message formats like JSON, unbounded data structures like domain lists, recursive control flows, and mutable session states, all of which are challenging (and sometimes impossible) to handle for automated symbolic provers like ProVerif and Tamarin. Furthermore, as the model grows to encompass the entire standard, it can be difficult to check that the model

itself is correct. For example, the model of TLS 1.3 in Tamarin [22] was over 2,000 lines, even though it did not cover some key protocol features like message formats and ciphersuite negotiation. For a model of this size, it is important to be able to execute and test it for specification errors, and to ensure that all the corner cases of the protocol are correctly exercised.

Hence, we write a detailed model of the ACME protocol in the F* programming language [53] which takes care of all the mentioned details. The full model is over 5,500 lines of F* and can be tested to generate symbolic traces for debugging. Given the level of detail in our model, it can be seen as a reference implementation of the ACME standard. Indeed, we show how our verified ACME client can be concretely executed to interoperate with real-world ACME servers, including those run by *Let's Encrypt*.

Symbolic Security Proofs for ACME in DY*. To formally prove security properties for our ACME model, we rely on a recent verification framework called DY* [11], a new approach for the symbolic security analysis of protocol code written in the F* programming language [53]. DY* is the latest in a line of works that uses dependent types to verify cryptographic protocols [3, 6, 8, 16–18, 32, 53]. The key difference with these prior works is that DY* explicitly encodes the global run-time semantics of distributed protocol executions in terms of a *global trace*, and the symbolic security analysis is proved sound with respect to this semantics *within* the verification framework itself. Using the global trace, it becomes possible to explicitly model protocol state, random number generation, and fine-grained compromise in a sound way without relying on external pen-and-paper proofs. This, in turn, allows for the verification of sophisticated security properties like forward secrecy and post-compromise security for cryptographic protocols like Signal [11].

Unlike automated symbolic provers like ProVerif or Tamarin that analyze *abstract* protocol models, the aim of DY* is to help programmers write and formally verify *executable* code that accounts for both the high-level design and the low-level implementation details of a cryptographic protocol and the application code that uses it. In particular, DY* supports the verification of recursive and stateful protocols with unbounded data structures like lists and trees, which are typically hard to reason about with automated provers that lack general induction. Conversely, unlike automated provers, proofs in DY* require manual annotation. For example, our 5,500 line ACME model requires a further 5,200 lines of proof, whereas the Tamarin and ProVerif analyses for 100-300 line models of small subsets of ACME are mostly automatic. However, the level of proof effort in our work is commensurate with other in-depth protocol analyses. For example, the mentioned 2,000 line TLS 1.3 model in Tamarin [22] required about 1,000 lines of lemmas, even for a high-level protocol model.

Importantly, the type-based methodology of DY* is modular and scales better than the whole-protocol analysis of automated provers. The verification of large protocols like TLS 1.3 in Tamarin and ProVerif can take tens of hours or even days of verification time, whereas in DY*, each module can be implemented and verified separately. For ACME we have 37 modules with an average verification time of about 1.5 minutes per module, totaling about 60 (single-core) minutes on a standard desktop.

DY*'s approach to enable such modularity is based on a combination of local and global reasoning: predicates on the global trace capture the inter-dependencies between different states and different components of the protocol while local invariants ensure that every component preserves these predicates. In our analysis, we state and mechanically verify the local invariants for each component, and then we prove that these local invariants, when combined with predicates on the global trace, yield the desired protocol security goals. Hence, our proof includes the composition of the protocol with every component, and if there were a composition flaw, the overall proof would fail.

Extensions to DY*. In principle, DY* is a suitable framework for the in-depth analysis of a large protocol standard like ACME. However, being a very recent analysis framework, DY* still has to demonstrate what it is really capable of. So far, DY* has not been used for such a large analysis, and we still needed to extend the DY* approach in three ways to model and verify ACME.

First, DY* assumes a standard symbolic equational theory for signatures, but as illustrated by an attack on an early draft of ACME presented in [2], many signature schemes are vulnerable to so-called key substitution attacks in which the adversary crafts a new verification key for a new message and the given signature. We therefore extend the model of signatures in DY* to account for such attacks (see Section 3.2) and use the resulting weaker assumptions on signatures when verifying our model of ACME.

Second, the domain authentication sub-protocol in ACME relies on a trusted channel between the domain owner and the ACME server. For example, the domain owner is expected to write a file to a web server that only she has access to, and we assume that the ACME server can securely read this file. A natural way to model such communication is via an authenticated channel, which is not currently supported by DY*. We extend the communication model of DY* to include an authenticated channel for each principal that only the principal can write to but anyone can read. This extension is completely generic, and hence, of independent interest.

Third, DY* has not been used to create interoperable implementations so far. We propose a novel approach to transform a DY* model into a concrete implementation that can successfully run protocol roles with real-world counterparts, and we use this methodology to test interoperability between our ACME client and other ACME servers.

Contributions. We present the first comprehensive formal model of the ACME standard (Section 6). Our 5,500 line model of ACME is written in the F* programming language and is executable. We illustrate its level of detail by demonstrating that our client code is, in fact, interoperable with real-world ACME servers, including *Let's Encrypt* (Section 8). We present symbolic security theorems of this model using the DY* framework: in particular, we prove that certificates are only issued to the rightful owner of the domains included in the certificate and that the overall protocol flow provides integrity guarantees to clients and servers (Section 7). Our proofs rely on novel extensions to the DY* framework that are of independent interest (Section 3). Our full proof development totals more than 16,000 lines of F* code and is one of the largest and most in-depth analyses of a cryptographic protocol standard in the literature.

2 DY*: SYMBOLIC SECURITY ANALYSIS IN F*

We here give a brief overview of DY* sufficient to follow the rest of the paper and refer the reader to [11] for details, to [13] for a tutorial-style introduction, and to [49] for more information on the umbrella project *REPROSEC*.

A Global Trace of Execution. DY* models the global interleaved execution of a set of protocol participants (or *principals*) as a trace of observable protocol actions (or *entries*). As a principal executes a role in some run of a protocol, it can send and receive messages, generate random values, log security events, and store and retrieve its session state, and each of these operations either reads from or extends the global trace.

In F* syntax, the global trace is encoded as an array of entries:

```
type entry =
| RandGen: p:principal → b:bytes → l:label → u:usage → entry
| SetState: p:principal → v:array nat → s:array bytes → entry
| Message: s:principal → r:principal → m:bytes → entry
| Event: p:principal → ev:string → ps:list bytes → entry
| Compromise: p:principal → sid:nat → version:nat → entry
type trace = array entry
```

Each trace index can be seen as a *timestamp* for the corresponding entry. An entry RandGen p b l u at index i indicates that a principal p generated a random value b at i with a secrecy label l and intended usage u (we discuss labels and usages later). The entry SetState p v s indicates that principal p stored an array of values s containing the current states of its active sessions (numbered $0..length(s)-1$), where the current version number of each session is recorded in v. The entry Message s r m says that a message m was sent on the network (ostensibly) by principal s for principal r. The entry Event p ev ps indicates that a principal p logged a security event ev with parameters ps. The dynamic compromise event Compromise p sid version says that the attacker has compromised a specific version of a session sid stored by principal p (and hence can obtain the corresponding session state).

Protocol Code in the DY Effect. The protocol code for each principal cannot directly read or write to the trace, but instead must use a typed trace API that enforces an append-only discipline on the global trace using a custom computational effect called DY. For example, the API provides a function gen for generating new random values with the following type:

```
val gen: p:principal → l:label → u:usage → DY bytes
  (requires (λ t0 → ⊤))
  (ensures (λ t0 r t1 → match r with
    | Error _ → t0 == t1
    | Success v → len t1 = len t0 + 1 ∧ entry_at (len t0) (RandGen p v l u)))
```

The pre-condition of the function (denoted by requires) is ⊤, indicating that it is satisfied in all input traces t0. The post-condition is stated in terms of the input trace t0, output trace t1 and the return value r. It says that if the function gen successfully returns (without an Error) then the trace has been extended by the corresponding RandGen entry; otherwise, the trace is unchanged.

Functions annotated with the DY effect are total (i.e., they always terminate) but they can return errors. They can call pure (side-effect free) functions like crypto primitives, or read entries from the trace, or add new entries at the end of the trace, but cannot do any other stateful operations. The DY* trace API offers a set of base functions in the DY effect that higher model layers (see below) build upon. In addition to gen above, it provides functions to send and

receive messages, store and retrieve states, and log security events. Using these functions, and a library of functions for cryptography and bytes manipulations, we can build stateful implementations of protocols like ACME, as we will see in later sections.

A Symbolic Cryptographic API. DY* provides a library for the manipulation of bytes. The interface of this library treats bytes as abstract using the type bytes and provides functions for creating constant bytes, concatenating and splitting bytes, and calling various cryptographic functions on bytes such as public-key encryption and signatures, symmetric encryption and message authentication codes, hashing, Diffie-Hellman, and key derivation. For example, the API for public-key signatures is as follows:

```
val vk: sk:bytes → bytes
val sign: sk:bytes → msg:bytes → bytes
val verify: pk:bytes → msg:bytes → sg:bytes → bool
```

The function vk computes the public verification key corresponding to a private signature key; sign creates a signature given a signing key and a message; verify checks a signature over a message using the verification key.

The library interface also provides a series of functional lemmas relating to these functions, stating, for example, that decryption is an inverse of encryption, or that splitting concatenated bytes returns its components, or (as depicted below) that signature verification always succeeds on a validly generated signature.

```
val verify_sign_lemma: sk:bytes → msg:bytes →
    Lemma (verify (vk sk) msg (sign sk msg) == true)
```

This cryptographic API is generic and can be implemented in many ways, including by calling concrete cryptographic libraries. DY* provides a *symbolic* implementation of bytes as an algebraic data type. Each function is either implemented as a constructor or a destructor of this type. For example, the function call sign sk msg is implemented using a constructor Sign sk msg of the bytes type, whereas verify is implemented by pattern matching over the input signature to inspect whether it has the form Sign sk msg for the corresponding signing key. This kind of symbolic encoding of cryptography is originally due to Needham and Schroeder [47], and forms the basis for all symbolic protocol analyses, including ProVerif [20], Tamarin [45], and prior refinement type systems [16].

Note, however, that this symbolic model is hidden behind the abstract cryptographic API, and hence is invisible to any protocol (or attacker) code that uses this library. Such code can only manipulate bytes by using the functions in the interface; it cannot, for example, extract the signing key from a signature or invert a hash function.

The Dolev-Yao Attacker. The symbolic Dolev-Yao active network attacker [26] is modeled as an F* program that is given full access to the cryptographic API and limited access to the global trace API. It can call functions to generate its own random values (by calling gen), send a message from any principal to any principal, and read any message from the trace. Notably, it cannot read random values or events from the trace, and *a priori* it cannot read the session states stored by any principal. However, the attacker is given a special function that it can call at any time to add an entry Compromise p sid version to the trace. Thereafter, the attacker can retrieve the compromised version of the state from the trace.

Of course, the attacker can also call any function in the cryptographic API using bytes it has already learned to compute new

messages that it can then insert into the trace, causing honest principals can receive them. Furthermore, each protocol model may provide the attacker with an additional API that it may use to initiate and control protocol sessions at both honest and compromised principals. The predicate attacker_knows_at index b says that the attacker can derive bytes b at a given index in the global trace, and hence it characterizes the attacker's knowledge.

Symbolic Debugging. Code written in DY* can be executed symbolically to obtain traces that can be printed and inspected for debugging. This kind of execution is invaluable to test the model and ensure that it behaves as expected. In particular, we can ensure that there isn't a bug in the protocol code that prevents protocol runs from finishing. Furthermore, we can write example attacker code and test potential attacks against our protocol implementation. Of course, the goal of verification is to ensure that no such attacker program can violate the intended security goals of the protocol.

Security Goals as Trace Properties. The security goals of each protocol are stated in terms of reachable global traces. To prove that some instance of bytes b is kept secret by a protocol, we ask if it is possible for the attacker to use some combination of calls to the crypto API, trace API, and the protocol API such that in the final trace t, the attacker knows b. Protocol authentication properties are stated in terms of correspondences between events logged at different principals. For example, we may ask that in all reachable traces, if a principal p ends a session with some peer principal p' by logging the event Event p "end"[p',x,...] at index i, then at some index j < i, the principal p' must have logged an event Event p' "begin"[p,x,...] with matching parameters. This way of encoding authentication properties as correspondences between events is similar to other symbolic analysis methods and is originally due to Woo and Lam [55].

The main proof methodology for proving security goals stated as trace properties is to establish an invariant over all reachable traces and prove that this invariant implies the desired goals. In particular, we need to prove that all functions that can modify the trace, either on behalf of honest protocol code or the attacker, preserve the invariant. Stating a complete global invariant that captures all protocol runs is challenging and time-consuming since a large part of the invariant typically involves generic assumptions about the well-formedness of bytes and the secrecy of keys. Hence, DY* offers a modular proof methodology, where programmers only need to define and prove local protocol-specific session invariants and security goals, and the framework fills in generic security invariants that are proved once-and-for-all for all protocols.

DY* defines a second layer of *labeled* APIs for cryptography and stateful code as well as a second computational effect DYL on top of the DY effect that enforces a global trace invariant called valid_trace. Functions in the trace API and with the DYL effect take valid_trace as both pre- and post-condition. The invariant consists of several components, some generic and some that have to be defined for each protocol. We describe these components below.

Secrecy Labels and Usage Predicates. The labeling predicate has_label i b l says that a bytes b has a *secrecy label* l at trace index i. Each bytes is assigned a unique label that indicates who can read it. For example, a label CanRead [P p1; P p2] indicates a secret that

only the principals p1 and p2 are allowed to read, whereas the label Public indicates that anyone can read it. Secrecy labels form a lattice, where can_flow i l1 l2 says that the label l1 is equal or less strict than the label l2 at trace index i. In particular, Public flows to all other labels, and CanRead [P p] can flow to Public at index i if Compromised p sid v occurs in the trace at or before i.

The labeled APIs enforce a labeling discipline that ensures that secret values never flow to public channels where the attacker can read them. In particular, the valid_trace invariant states that all network messages must have the label Public, and all states stored at a principal p must flow to CanRead [P p]. If a secret value has to be sent over the network, it must first be encrypted with a key whose label is at least as strict as the message's label. We refer the reader to [12] for the full set of labeling rules.

In addition to secrecy labels, the labeled APIs also enforce usage pre-conditions. Each key is assigned an intended usage, so for example, a signature key cannot be used as an encryption key. Furthermore, each key is assigned a usage predicate controlling what kinds of messages it may be used to encrypt or sign. Of course, these restrictions only apply to honest principals. For example, the labeled API for the signature and verification functions is as follows:

```
val vk: i:nat → p:principal →
        sign_key i (CanRead [P p]) → verify_key i (CanRead [P p])
val sign_pred: i:nat → p:principal → msg:bytes → pred
val sign: i:nat → p:principal → sign_key i (CanRead [P p]) →
          m:msg i Public{sign_pred i p m} → msg i Public
val verify: i:nat → p:principal → verify_key i (CanRead [P p]) → m:msg i Public →
            sig:msg i Public → b:bool{b ⟹ (compromised i p ∨ sign_pred i p m)}
```

In this API, signature keys are now labeled with a secrecy label CanRead [P p] corresponding to some principal p. The corresponding verification keys are Public, and we additionally annotate them with the label of the corresponding signature key. The predicate sign_pred is a usage predicate that each protocol defines to indicate what kinds of messages may be signed in the course of the protocol. This predicate is then used as a pre-condition for sign, ensuring that protocol code does not accidentally call sign with a message that does not conform to sign_pred. Conversely, if verify succeeds then the API guarantees that the signature must be valid, and hence the signed message must satisfy sign_pred, unless the signature key was compromised by the attacker.

Protocol State Invariants and Security Goals. Using secrecy labels and usage predicates, the labeled APIs ensure that the only messages that may be sent out on the network obey the labeling discipline. In addition, each protocol defines a state invariant state_inv p v s that must hold for each state that a principal tries to store by adding an entry SetState p v s into the global trace. This invariant typically records the secrecy labels of all the values stored in the state as well as any integrity properties known about these values at the current stage of the protocol. In particular, the invariant requires that the labels of all session states stored at p must flow to CanRead [P p], that is, they must be readable by p.

To state the secrecy goals of a protocol, it is enough to annotate a desired protocol value with a secrecy label and typecheck the protocol code against the labeled APIs. For well-labeled programs, DY* provides a secrecy lemma that states that bytes with a label (say) CanRead [P p1; P p2] cannot be learned by the adversary unless p1

or p2 are compromised. To state authentication goals as event correspondences, we define an event pre-condition event_pred p ev ps that states when a principal p can issue Event p ev ps. For example, we can ask that an event end [x] can only occur in a trace where begin [x] has already occurred before. By typechecking, we ensure that this predicate holds at each logged event, and hence that the protocol authentication goal is preserved.

3 EXTENSIONS OF DY*

We here describe our generic extensions to DY*. We will later use these extensions for our analysis of ACME, but all of these extensions are of independent interest.

3.1 Authenticated Channels

So far, DY* supports public messages only without any integrity guarantees, i.e., the overall communication model assumes an adversary who controls the network. Some protocols, such as ACME, assume as a setup assumption that principals can exchange certain messages in an authenticated way. If a party receives such an authenticated message, it has the guarantee that the sender claimed in the message indeed sent the message and it has not been modified in transport – jumping ahead, our formalization is much broader than this. Note that an adversary can still block authenticated messages and learn the content of such a message.

To add authenticated messages to DY*, we extend the basic communication model as sketched in what follows.

```
type entry = |... | AuthMessage: s:principal → r:principal → m:bytes → entry
val auth_send_pred: idx:nat → s:principal → r:principal → m:bytes → Type0
val auth_send: idx:nat → s:principal → r:principal → m:msg idx public → DYL nat
    (requires (λ t0 → ... ∧ auth_send_pred (len t0) s r m))
    (ensures (λ t0 now t1 → ... ∧ is_auth_message_sent_at now s r m))
val auth_receive_i: idx:nat → r:principal →
    DYL (now:nat & s:principal & msg now public)
    (requires (λ t0 → ...))
    (ensures (λ t1 (|now,s,m|) → ... ∧ (∃ r'.
             is_auth_message_sent_at idx s r' m ∧
             (compromised idx s ∨ auth_send_pred idx s r m))))
let valid_trace (tr:trace) = ... ∧
  (∀ (idx:nat) (s r:principal) (msg:bytes). i < len tr ⟹
          (entry_at idx (AuthMessage s r msg) ⟹
             (compromised idx s ∨ auth_send_pred idx s r msg)))
```

We first have to extend the type for trace entries to include entries for authenticated messages. These contain the sender (s), the receiver (r), and the message (m). More importantly, to provide guarantees for receivers of authenticated messages, we introduce a predicate auth_send_pred. This predicate states properties of an (authenticated) message sent by the sender to the receiver at trace index idx. In a verification process, when the sender sends an authenticated message, one has to prove that this message actually satisfies the predicate. In fact, the auth_send function can only be called by an honest sender if the message it wants to put on the global trace satisfies this predicate (see the requires clause of that function). Conversely, when a party receives such a message, it can be sure that the predicate is satisfied or the sender is corrupted (see the ensures clause of the receive function). Finally, we extend the valid_trace predicate such that for every authenticated message, either the (ostensible) sender is corrupted or auth_send_pred holds.

We point out that the concept introduced here greatly generalizes the basic notion of an authenticated channel. We cannot only express that a message has been sent by a certain sender, we can

even express that this message has certain properties and, for example, that the (honest) sender, when sending this message, was in a certain state. The authenticated send predicate can be customized per application, and hence, can precisely reflect relevant aspects of the behavior of honest parties in that application.

3.2 Modeling Key Substitution Attacks

Key substitution attacks [46] (also called duplicate signature key selection [19] or (no) exclusive ownership attacks [48]) are subtle properties of signature schemes that are not excluded by standard cryptographic security definitions for signatures, and that can lead to attacks on protocols, with an early draft of ACME being one example [2] (see also Section 5). However, the default model of signatures in DY* does not account for such attacks.

We extend the algebraic model of signatures in DY* to allow key substitution attacks. More specifically, following [36] we formulate *(no) destructive exclusive ownership (DEO)* and *(no) conservative exclusive ownership (CEO)*, with CEO being a special case of DEO.

Roughly speaking, if a signature scheme fails to provide DEO, given a signature s on a message m, an attacker can forge a new verification key for a message m' (possibly $\neq m$) such that s verifies under the new key for m'. Similarly to [36], we specify a function that enables the adversary to generate such a verification key:

```
val deo_gen: m':bytes → sig:bytes → bytes
```

This function is used to generate a new type of DY* bytes:

```
type bytes = ... | DEOgen: m':bytes → sig:bytes → bytes | ...
```

We extend the verify function for signatures such that if the verification key provided to the function is of this type, the verification succeeds if the (forged) key was generated for the signature and message.[1] We verify that the modified function indeed reflects no-DEO using the following lemma, which expresses this property:

```
val deo_gen_verify_lemma: m:bytes → m':bytes → sk:bytes →
    Lemma (ensures (verify (pk (deo_gen m' (sign sk m))) m' (sign sk m) == true))
```

This lemma is automatically proven by F*'s type checker. DY* contains similar lemmas to ensure that signatures work as intended. These are unaffected by our change but illustrate that our extension does not break the existing semantics of signatures. There is, however, one lemma that (obviously) fails under this modification.

```
val sign_verify_lemma: vk:bytes → msg:bytes → tag:bytes →
    Lemma (verify vk msg tag ⟹
        (∃ sk msg' . tag == sign sk msg' ∧ vk = pk sk ∧ msg = msg'))
```

This lemma states that any signature must have been created with the correct key, which, however, is not the case anymore. We therefore remove this lemma from the (extended) DY* framework.

No-CEO is a special case of no-DEO where $m' = m$ (see, e.g., [48]). We, hence, can formalize no-CEO similarly to no-DEO.

3.3 Our Approach for Interoperability

As mentioned in Section 2, DY* allows for symbolic debugging of model code and enables us to execute and test our model as a sanity check. Still, there might be subtle mismatches between the model and the real-world specification. Therefore, we want to test a DY* model against a real-world implementation, i.e., we take
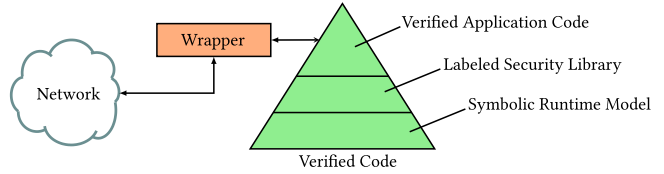
Figure 1: Overview of the interoperable implementation architecture. The core of the application (on the right) consists of the verified F* code.

the symbolic implementation of one party, connect it to a real network, and run it with real-world counterparts. This way, we check for interoperability to strengthen confidence in the model and to illustrate that the model is as precise as a reference implementation.

To this end, we leverage the symbolic debugging feature of DY* to compile our verified model (including the DY* framework) to OCaml code. We further implement a wrapper in OCaml that connects the (now) executable model with the real world (see Figure 1). This wrapper mainly translates messages from real bytes to symbolic bytes and vice-versa. For this purpose, the wrapper maintains a mapping between symbolic nonces and keys to concrete counterparts. Based on the message it receives from the real network, the wrapper also selects the protocol endpoint to call.[2] Hence, some parts of the wrapper are application-specific. But it does not perform deeper application logic or state management. In particular, we do not change the code generated from the DY* model. It is this code, with all its layers, that is called by the wrapper to actually carry out the protocol. The code, for instance, still maintains the trace as a (now local) data structure to keep state. We illustrate our approach in Section 8, with ACME as an example.

We note that our approach for interoperability does not aim to yield a fully verified real-world executable but rather allows us to perform in-depth functional tests of the entire (verified) model with real-world counterparts.

4 THE ACME PROTOCOL

The ACME standard defines a protocol between a client and a server. In practice, a web server that wants to get a new certificate acts as an ACME client, and a CA acts as an ACME server. To request a certificate, the ACME client creates an *order* for one or more of its own domains and sends the order to the ACME server. The ACME server then responds with an *authorization* message containing one or more *challenges* (usually a nonce) for each of the domains. The client publishes each challenge on (the HTTP web server or the DNS server of) its corresponding domain (along with some other data, see below) to prove that the client indeed controls the domains. The ACME server verifies that the challenges have indeed been published accordingly for all requested domains. Finally, the client creates a key pair for the domains and provides the ACME server with the public key (contained in an X.509 *certificate signing request (CSR)*). The ACME server then issues a certificate containing the public key and all domains previously requested by the client.

---

[1]Note that our symbolic model allows privileged functions such as verify to "unwrap" signatures to inspect the message they have been created for.

[2]Note that this way, we leave scheduling of our application to the network, which is a standard assumption for protocol analyses.
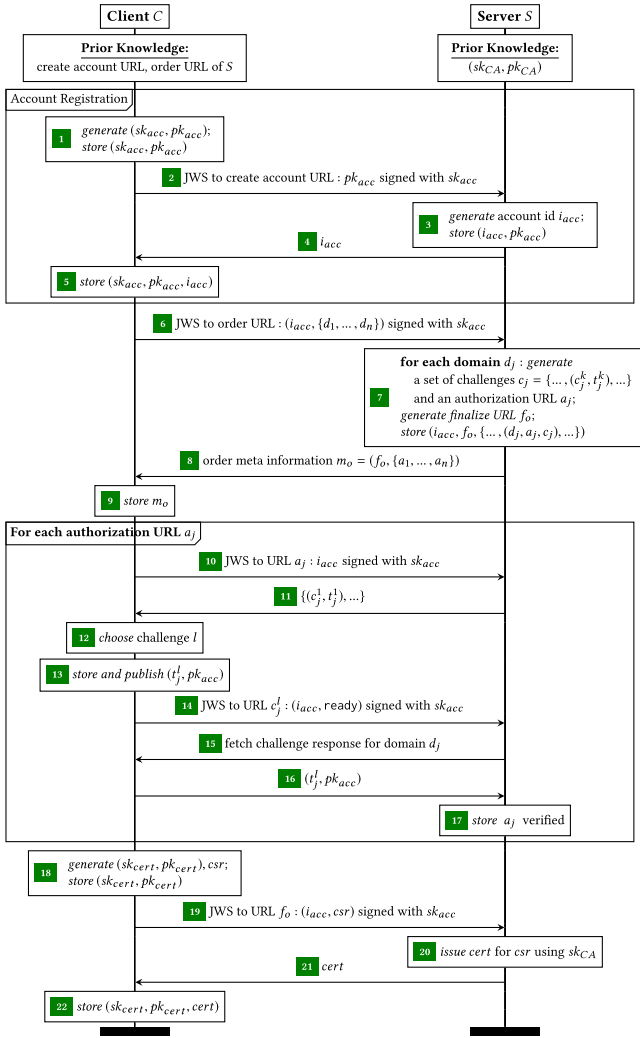
Figure 2: ACME protocol flow. We note that this figure simplifies several aspects of ACME for brevity, e.g., the presentation of JWS messages, which use a well-structured data format and include the URL they are sent to. JWS messages have to be signed by the sender, and this signature is verified by the receiver. Also, the last step has been simplified. In reality, the server does not return the certificate in Step 21, but a URL instead, at which the client then retrieves the certificate. In our model, however, we include all such details.

The actual ACME protocol is designed to work asynchronously and breaks the sketched flow into several (smaller) steps. Each of these steps is linked to the flow using unique identifiers (provided in URLs) that are issued by the ACME server step-by-step during the flow. The protocol requires the client to authenticate each message (based on the *JSON Web Signature (JWS)* standard [37]).

4.1 Main Protocol Flow

Figure 2 depicts a typical ACME protocol flow (simplified for presentation). Before the client actually orders a certificate, it first

creates an asymmetric key pair (pk_{acc}, sk_{acc}) in Step ① that will be used to sign (and verify) protocol messages sent by the client to the ACME server. The client registers pk_{acc} at the server ②, which in turn creates an account identified by some URL[3] i_{acc} for this key ③ and sends it to the client ④. The client stores the identifier along with the corresponding key pair ⑤, and henceforth, each request to the server includes i_{acc} and is signed with the signing key sk_{acc}.

To request a new certificate, the client creates an *order* for a set of domains (belonging to the client) ⑥. For each of these domains, the server generates a so-called *authorization* object that tracks the ownership verification of the domain ⑦. Each authorization is identified by a URL a_j and includes one or more *challenges*, one for each verification method that the server supports. Each challenge is identified by some URL c_j^k, which is later used to state which challenge the client decided to solve. For each challenge, the server also generates a nonce t_j^k (see below). The server further generates a finalize URL f_o for the order which the client uses at the end of the flow to request the actual certificate. After the order has been created at the server, the server provides the client with meta-information about the order ⑧, including the authorization URLs at which the client can retrieve the challenges for each domain as well as the finalize URL, all of which the client needs to know to perform later steps of the protocol ⑨.

To prove ownership of each domain, the client first retrieves the respective authorization object (including the challenges) from the server (Steps ⑩ and ⑪). To make an authorization *valid*, i.e., convince the server that the client indeed owns a domain, the client can choose one of the challenges ⑫ (say challenge l) as it is sufficient for ACME to solve only one challenge (recall that the set of challenges represents different verification methods). Typically, a challenge is to publish the nonce t_j^l (concatenated with pk_{acc}) in some way that only someone with a high level of control over that domain is able to do ⑬.[4] To trigger verification of the challenge l that the client has chosen, the client sends a request to the URL c_j^l ⑭. The ACME server now checks whether the client was able to publish the nonce correctly. To this end, the server sends an HTTP request to some well-known path of the respective domain ⑮. The server verifies that the HTTP response ⑯ sent by the (web server of) the client contains (t_j^l, pk_{acc}) and records that the authorization has successfully been verified ⑰. Note that the ACME standard assumes that an attacker cannot spoof or manipulate the verification response ⑯.[5] Otherwise, an attacker can trivially attack ACME and verify ownership of arbitrary domains. In our model (see below), we use an authenticated channel to transfer the message of Step ⑯ (all other messages are sent over an insecure channel). After the ACME server successfully verified each authorization, the server considers the order of the certificate to be valid and allows the client to request a certificate. To this end, the client creates a fresh

---

[3]The account URL not only identifies this account but also allows the client to send requests to the server to change account metadata, replace the account key, or deactivate the account.

[4]Currently, there exist three challenge types, http-01, dns-01, and tls-alpn-01 (the latter is specified in a separate standard [50]). All challenge types work very similar, and for presentation, we here assume that challenge l is an HTTP-01 challenge.

[5]In practice, CAs verify the publication of the challenge data several times using different network routes to minimize the probability of an attacker being man-in-the-middle for all verification messages, see, e.g., [23]. See also Section 10.2 of [5].

key pair along with a corresponding X.509 CSR ⎡18⎤, and transfers the CSR to the server ⎡19⎤. The ACME server checks whether the CSR lists the same set of domains as the order and (if the check succeeds) issues the certificate for this CSR ⎡20⎤ and provides the certificate to the client ⎡21⎤ which in turn stores the certificate for usage ⎡22⎤. We note that for brevity of presentation, the protocol description above focuses on the main functionality of ACME, and in particular, does not present mechanisms for replay protection, account management, pre-authorization, and certificate revocation (see Sections 6.5, 7.3, 7.4.1 and 7.6 of [5]).

4.2 Informal Security Goals

Informally, the ACME protocol intends to ensure that clients can obtain certificates only for domains under their control. The security considerations section of the ACME specification (see Section 10 of [5]) divides this goal into two concrete statements, intending to restrict the authorizations to the rightful parties: (1) A client should only be able to get authorizations for domains it controls, and (2) authorizations for a domain associated with an account (via the account public key) should not be used by other accounts. Without either of these properties holding true, an attacker could get certificates for someone else's domains.

5 ATTACKS AND VULNERABILITIES

The previous section illustrates that the 100 page ACME standard [5] forms a quite complex, asynchronous protocol, which consists of unbounded recursive control flows, complex state management, and messages with unbounded data structures, such as lists or trees, which themselves again need to be processed in loops.

There can be potential flaws and problems on various levels, including logical and implementation flaws as well as ambiguities in the RFC that programmers might interpret incorrectly.

For example, in an early draft of the standard, the value published for domain ownership verification (see Steps ⎡13⎤ and ⎡16⎤ of Figure 2) used to be $\sigma = \text{sign}(t_j^l, sk_{acc})$ instead of (t_j^l, pk_{acc}), i.e., the client published only the value of a signature on the respective domain challenge t_j^l.[6] As ACME allows clients to use signature schemes, which do not provide DEO, such as RSA-PSS and certain instantiations of ECDSA, an adversary can easily carry out an attack to obtain a certificate for a domain of some honest client [2]: After the honest client has published σ for her domain d, the adversary uses its own account to start another ACME flow for d. After the ACME server has issued a new challenge t' (to the attacker) to verify ownership of d, the adversary forges a new key pair for its account such that σ is a valid signature for the challenge t' under the attacker's new public account key.

One of the most important implementations of an ACME server, *Boulder*, contained an implementation error in the ownership verification loop for domains which resulted in Boulder checking only the first domain in the list of an order, allowing an adversary to obtain certificates for domains it does not own [35]. As already mentioned in the introduction, as a result of this error, Let's Encrypt had to revoke more than 3 million certificates.

---

[6]Note that at this time, ACME used different challenge concepts for the several domain verification sub-protocols. This verification mechanism was part of the DNS-based authorization mechanism.

	# modules	fLoC	pLoC	Σ LoC	Time
DY*	13	2,180	3,806	5,986	6 min
ACME model	37	5,531	5,191	10,722	61 min
Σ	50	7,711	8,997	16,708	67 min

Table 1: Overview on lines of code and verification time, including number of modules, functional lines of code (fLoC), lines of code for proofs and security property (pLoC), and the verification time using the F* type checker on an off-the-shelf machine (single-core on Intel Xeon E5-2620 v4 @ 2.1 GHz). Note that verification in DY* is modular; each module is verified separately, the *verification time per module is less than 1.5 minutes on average.*

An example of an ambiguity in the ACME RFC is that the standard does not clearly state that the ACME flow must be performed using the same account, i.e., the server must check whether all requests sent by the client are signed with the same key within one flow. While it is obvious that the server must perform this check, the standard only suggests this quite implicitly. Developers who just "code down" the standard might easily miss this check.

As also demonstrated by other real-world protocols, flaws can be quite subtle and be rooted in both logical protocol flaws as well as low-level mechanisms and ambiguities, such as state management [54], control flow decisions [9], insufficient checks [29], and incomplete or ambiguous data structures [30, 51, 52].

For critical protocols, such as ACME, which affect the security of the whole Internet infrastructure, it is therefore important to carry out a very detailed analysis by rigorously following the specification and along the way also clarify potential ambiguities. This results in strong and meaningful security guarantees of the standard, and our modeling/implementation artifact can serve as a reference implementation alongside the standard itself to guide programmers.

6 DY* IMPLEMENTATION OF ACME

We modeled ACME clients and servers as F* programs built on top of DY*'s labeled security API. Our model supports an unbounded number of principals, where each principal can own any number of domains, and can play the role of a client or server (or both).

Our model covers most of the ACME standard, including all steps specified by ACME to order, verify, issue, and retrieve a certificate, as well as a detailed specification of all relevant data structures. Moreover, our model also covers semantics of underlying protocols such as HTTP requests and responses, including HTTP headers and ACME's special construction of POST-as-GET requests. This results in a comprehensive model comprising about 5,500 lines of functional F* code alone, *excluding* the DY* framework, comments, white space, and lemmas and proofs for the security properties (see Table 1). To further demonstrate our coverage, we refer to Appendix C where we map sections of the ACME specification [5] to F* modules in our model.

Modeling Decisions. To ease analysis, our model leaves out certain security mechanisms required by ACME, and by this, safely over-approximates them. That is, we omit the usage of TLS, i.e.,

ACME messages are exchanged unprotected (e.g., no confidentiality or integrity) on the network. The only exception is that the verification response (Step 16 of Figure 2) is sent over an authenticated channel, matching the key assumption of the ACME standard, as explained in Section 4.1. Further, our modeled server does not enforce ACME's replay protection mechanism (DoS protection), which, as shown by our analysis, is not necessary for ACME's central security properties. Moreover, in the case of verification mismatches, our implementation does not output a terminating error message but instead aborts only the local execution of the respective protocol step without updating the state machine, allowing the same protocol flow to be continued, and hence, leaving the adversary further options. Following the same reasoning, we also do not model expiration of ACME orders, authorizations, and challenges.

6.1 Application-specific Data Structures

In our model, we closely follow the specification, including real-world message formats. This is security-critical. For example, we capture that the ACME server points the client to URLs for subsequent steps, which can potentially lead the client to interact with an unintended ACME server in the same protocol flow. Modeling these detailed message flows directly in terms of DY* bytes would be too cumbersome. Instead, we define ACME-specific high-level data structures as F* record types and implement functions to convert these high-level data structures to bytes and back. Using F*'s theorem prover, we prove that these conversion functions are inverses of each other (see Appendix A for a full example). This approach provides multiple benefits over a bytes-only modeling. First, by relying on the F* type checker to verify initialization of all record fields with values of the correct type, we avoid all modeling errors related to inconsistent message serialization and parsing. Second, as data structures can be modeled very close to their specification, writing and (manually) comparing the model against the standard becomes much easier. We believe this modeling approach is generally useful, beyond ACME, for protocols with complex messages.

As an example, we show our high-level data structure for an ACME order and compare it with its specification in the standard:

```
type acme_order = {
  status: option // optional because new order messages (7.4 of [5]) do not have a status
    (s:acme_status{s=Pending || s=Ready || s=Processing || s=Valid || s=Invalid});
  identifiers: doms:(Seq.seq domain){Seq.length doms > 0};
  authorizations: option (Seq.seq url);
  finalize: option url; // Optional for the same reason as status
  certificate: option url}
```

This defines an F* record type acme_order with five fields. E.g., the field identifiers holds a value of type Seq.seq domain, i.e., an unbounded sequence of domains, with an additional refinement stating that this sequence cannot be empty. (Identifiers are shortened for presentation, see module ACME.Data in [10] for full details.)

Comparing this type definition with the following excerpt from Section 7.1.3 of the ACME specification [5], it is easy to see that our model closely resembles the specification:

```
status (required, string): [...] Possible values are "pending",
  "ready", "processing", "valid", and "invalid". [...]
identifiers (required, array of object): An array of identifier
  objects that the order pertains to. [...]
authorizations (required, array of string): [...] Each entry is
  a URL from which an authorization can be fetched [...].
finalize (required, string): A URL that a CSR must be POSTed to
```

```
  once all of the order's authorizations are satisfied [...]
certificate (optional, string):  A URL for the certificate that
  has been issued in response to this order.
```

6.2 ACME Client & Server APIs

The core of our ACME model is a set of APIs implementing the ACME protocol steps on both client and server side. These functions use the DYL effect provided by DY*, which allows us to make use of DY*'s labeled application API, e.g., to send and receive messages (recall that DYL implicitly carries the global execution trace and requires users to retain the valid_trace invariant). We here present one such function to illustrate our modeling, namely, the ACME server endpoint to place a new order. It has the following type:

```
val acme_server_new_order_nw:
    acme_server_principal:principal → // identity of the server
    msg_idx:nat → // trace index of send(http request)
  DYL nat // trace index of send(http response)
    (requires (λ t0 → ⊤)) // No precondition (except for valid_trace)
    (ensures (λ t0 result t1 → ⊤)) // No post condition (except for valid_trace)
```

This function takes the name of a server principal and an index (msg_idx) in the global trace at which it expects to find a message carrying an HTTP request containing an ACME new order message (as specified in Section 7.4 of [5]), i.e., an instance of the acme_order type presented above. The function either returns an error (e.g. if the input message is malformed) or it adds an HTTP response to the trace and returns an index to this message in the trace.

The function acme_server_new_order_nw is implemented as shown in Figure 3. Note that this code is roughly what one would expect from a "regular" ACME implementation in a functional programming language, e.g., the received message undergoes several parsing steps. The code precisely models authorization objects as specified in [5]: an acme_order does not contain the authorization objects themselves, but URLs from which the ACME client has to retrieve the actual authorization objects, a distinction that is security-critical for ACME.

6.3 Functional Tests and Attacks

To test the functional correctness of our ACME model, we use DY*'s symbolic debugging capabilities (see Section 2). Of course, as outlined above, our ACME model itself comprises a set of functions representing protocol steps. Thus, to actually run the protocol, we need to implement an entry point which calls these protocol step functions. We implement several such scheduling functions, modeling different runs of ACME, allowing us to: 1) Empirically verify that our models for ACME clients and servers are functionally correct in the sense that they can finish the protocol with each other. 2) Print and inspect the resulting symbolic traces produced by these protocol runs. For example, we have encoded a (benign) run similar to Figure 2 and checked whether the client indeed stores the correct certificate in the end. 3) Check whether protocol steps fail as expected with a faulty/malicious scheduler, i.e., an attacker. For example, we implement a run where a malicious ACME client orders a certificate for a set of domains, one of which it does not own, and we check that the protocol cannot continue (as expected) when the client is asked to prove control over that domain.

Of course, these tests do not prove security (see Section 7), but they serve to increase confidence in the functional correctness of

```
// Note: All parse functions throw an error if parsing fails
let acme_server_new_order_nw server msg_idx =
  // receive_i ∈ DY★, "sender of message" is untrusted information
  let (sender_of_message, message) = receive_i msg_idx server in
  let http_req = parse_http_request message in
  if (http_request_header_contains_domain_of_server http_req server) then
    let http_resp = acme_server_new_order_http server http_req in // See below
    let http_resp_bytes = serialize_http_response http_resp in
    send server sender_of_message http_resp_bytes
  else
    error "Request_was_sent_to_wrong_server"

let acme_server_new_order_http server http_request =
  let jws_request = parse_jws_acme_request http_request.req_body in
  if not(dfst (verify_jws_acme_request http_request server jws_request)) then
    error "Invalid_signature_or_invalid_request_JWS"
  else
    let order_object = parse_acme_order jws_request.payload in
    let ((UserAccount _acc_pub_key acc_url), account_session_id) =
      retrieve_account_session_for_kid server jws_request.key_id in
    let server_domain = get_dom_from_host_header http_request.req_headers in
    let (updated_order, authzs) =
      acme_server_new_order_helper order_object server_domain in // See below
    let new_order_session_idx = store_order_object_in_server_state
      server updated_order account_session_id acc_url acc_pub_key in
    store_authzs_to_server_state server authzs 0 new_order_session_idx;
    // Return an HTTP response containing the updated order. req_id allows relating this
    // response to the corresponding request (modeling the TCP connection beneath HTTP).
    new_order_create_response http_request.req_id updated_order

let acme_server_new_order_helper (order_obj:acme_order) (server_domain:domain)
  : DYL ( // Returns a tuple (order, authorizations)
    order:acme_order{(order.acme_order_status = Some Pending)} ×
    Seq.seq acme_authorization
  ) =
  // guid_gen ∈ DY★ generates a unique, but not necessarily secret value
  let finalize_path_nonce = guid_gen () in
  let finalize_url = create_finalize_url server_domain finalize_path_nonce in
  // create a sequence of authorization objects (Sec. 7.1.4 & 7.4 in ACME specification)
  let authzs = create_authorization_objects server_domain order_obj.identifiers 0 in
  let authz_urls = extract_urls_from_authzs authzs 0 in
  let updated_acme_order = {
      status = Some Pending;
      identifiers = order_obj.identifiers;
      authorizations = Some authz_urls;
      finalize = Some finalize_url;
      certificate = None
  } in
  (updated_acme_order, authzs)
```

Figure 3: Implementation of our ACME server's *new order* API. For brevity, we omit most annotations for F★'s theorem prover as well as some helper functions and technicalities. See module `ACME.Server.NewOrder` in [10] for full details.

our model. We, for example, found and fixed an error in our server model which prevented the server from accepting (valid) requests. We note that finding such errors is much harder in non-executable symbolic models, especially if the model is as detailed as ours.

In addition, we can also use our test framework to try out attacks based on known implementation flaws. For example, we extended the server API with a faulty function, resembling the *Boulder* flaw [35] (see Section 5), and demonstrated, by writing a new scheduler function, that an adversary can indeed exploit this bug to obtain a certificate for a domain it does not own.

6.4 Modeling Limitations

While our ACME model is quite comprehensive and detailed, there are some limitations: 1) We focused on the ACME protocol and

thus omitted certification authority aspects, in particular certificate revocation and account creation – we over-approximate the latter by assuming that an arbitrary number of honest and dishonest accounts has been initialized. 2) The ACME specification itself only specifies *DNS identifiers*, i.e., domains, but allows extensions to define other identifier types, e.g., email addresses. Our model does not include that extension point. 3) The ACME specification defines an optional feature called *pre-authorization* (see Section 7.4.1 of [5]), allowing clients to finish an ACME authorization independent of a concrete order. Since this is an optional feature and even Let's Encrypt does not support it [39], we chose not to model it. 4) As we do not model TLS (see above), we modeled DNS as a trusted function mapping domains to principals. Note that this does not impede the attacker's capabilities at all: Due to the lack of TLS, the attacker can redirect network messages without the receiver noticing. Finally, of the two (structurally similar) challenge types defined in the ACME specification, dns-01 and http-01, our client and server only implement the latter. Note that this does not impede compliance with the specification as support for either challenge type is optional.

7 SECURITY PROPERTIES AND PROOF

Our analysis considers several security properties: The central, most obvious property for certificate issuance is the *secure binding property*, which guarantees that an honest ACME server only issues certificates to the rightful owner of the domains stated in the respective certificate. This property not only captures the security goals of ACME (see Section 4.2) but is even stronger, as it holistically captures the overall issuance process. Moreover, we also state and prove *integrity properties* for the entire ACME flow, by which we go beyond the security goals stated in the specification as well.

We emphasize that our analysis does not consider just the ideal protocol flow as depicted in Figure 2 but considers an unbounded number of ACME servers and clients that can perform an unbounded number of (interleaved) protocol sessions in parallel. Furthermore, we consider an active network attacker that can (nondeterministically) intercept, block, and spoof messages, freely schedule the execution of the protocol, and corrupt any principal at any time (as discussed in Section 2).

7.1 Secure Binding Property

The secure binding property states that if an honest ACME server issues a certificate for a set of domains where at least one domain in that set is owned by an honest client, then the private key corresponding to the public key bound in the certificate is unknown to the adversary. This way, we capture that (as an outcome of every possible run of ACME) the attacker can never obtain a certificate that enables it to impersonate someone honests's domain. More formally, we define this property as follows:

DEFINITION 1. *Let t be a global trace of the DY★ model of ACME, s be an honest principal (an ACME server) in t, D be a set of domains, k be some private key, and cert some certificate issued by s in t for the set D and the public key corresponding to k. We say that t fulfills ACME's secure binding property for s, D, k, and cert if (in t) k is not derivable by the adversary or all domains d ∈ D are owned by dishonest principals (the adversary).*

Based on this definition, we state the following theorem:

THEOREM 1. *For all possible global traces t of the DY* model of ACME and s, D, k, cert as above, t fulfills ACME's secure binding property for s, D, k, and cert.*

We highlight that by proving this theorem for our detailed ACME model, we can exclude many classes of attacks. In particular, we can exclude the attacks described in Section 5, e.g., attacks based on logical errors in the protocol flow, ambiguities in the specification (e.g., not stating explicitly that the ACME server must enforce that the same account is used in one protocol flow) or even potential implementation glitches unintentionally implied by the specification. We state (and prove) the above theorem as a lemma in DY* which requires that the secure binding property follows from the valid_trace invariant. We discuss this invariant in the proof structure paragraph below (see also Section 2 for the general proof technique in DY*). We formulate this theorem in DY* as follows (slightly simplified for presentation):

```
val secure_binding_theorem:
    s:principal → // any (server) principal
    trace_idx:nat → // any trace index
    cert:certificate → // any certificate
    dom:domain → // any domain
    priv_key:bytes → // any private key
    DY unit
    (requires (λ t0 → ( valid_trace t0 ∧ // any trace t0 that is valid
        // if the server s updated its internal state at trace index trace_idx
        // and stored the certificate cert in a state entry of this updated state
        is_certificate_in_server_state s trace_idx cert ∧
        // if the domain dom is one of the domains for which the certificate is issued
        is_domain_in_certificate cert dom ∧
        // if the key priv_key is the private key to the certificate public key
        pk priv_key == cert.pub_key ∧
        // and if the server is not corrupted by the attacker
        ¬(is_principal_corrupted_before trace_idx s))))
    (ensures (λ t0 _t1 → (t0 == t1 ∧ (
            // then the attacker cannot derive the private key priv_key
            is_unknown_to_attacker_at (len t0) priv_key ∨
            // or the principal that owns the domain is corrupted
            is_principal_corrupted (owner_of_domain dom)))))
```

Proof Structure. In the following, we give a simplified high-level overview of how we prove the secure binding property within DY*. We provide the full proof of the property in our supplementary material [10] and only point to selected modules in what follows.

As already stated, the theorem must be implied by the valid_trace invariant. Recall that valid_trace includes, among others, the state invariant of the principal states state_inv and the authenticated send predicate auth_send_pred (see Sections 2 and 3.1). We need to construct the components of valid_trace such that they reflect relevant aspects from the different stages of the protocol. The majority of the work is then to ensure that valid_trace is preserved throughout all modules of the model.

One central property inferred from the challenge verification step is the connection between the owner of a domain and an ACME client account. When the ACME server receives a verification response containing the account's public key (see Step 16 of Figure 2) via an authenticated channel, we can infer that, if the sender is an honest principal, the sender of the message indeed owns the private key to the account.[7] We express this fact in auth_send_pred. Of course, the client must obey this predicate when sending the verification message, verified by F*'s typecheck.

We propagate the connection between the domain owner and the account key using DY*'s labeling system and encode this fact in the state invariant of the ACME server.[8] More precisely, if the server sets the status of an authorization to valid, then the account key related to the authorization must be labeled with the domain owner. Every time the server updates its state, we must ensure that this invariant holds true. In particular, the server's function processing the verification response infers this property from the guarantees of the authenticated channel.[9]

To connect the issuance of a certificate to the domain verification, we have to enrich the state invariant further: Whenever the server issues a certificate (as recorded in the server state), we require that the certificate key is labeled such that it belongs to the rightful domain owner for the domains in the certificate.[10] To prove that the server preserves this invariant as well, we first link the processing of the message containing a CSR to the guarantees of the ownership verification sketched above: Recall that, whenever the server receives a CSR for an ACME order to issue a certificate, the server checks whether the account key used for signing the JWS containing the CSR is the same key associated with the ACME order (as recorded in the server state). Following from the state invariant constructed above, we have that the JWS containing the CSR must have been signed by the rightful owner of the domains. Next, we link the label of the key contained in the CSR (that will be the key of the certificate) to the owner of the domains: To this end, we define the application-specific signature predicate such that it provides the guarantee that honest principals only ever create a JWS with a CSR if they own the key contained in the CSR. We again convey this information using the labeling system.[11]

Finally, to prove the secure binding theorem, we use the invariants on labels of certificate keys as sketched above and the generic secrecy properties of DY*'s labeling system.[12]

Modularity. We highlight that the DY* framework enables the proofs to be highly modular, illustrated by the example above, where the server needs to fulfill specific properties before being able to store the certificate. Overall, this leads to quite local proofs for each function, making the proof very efficient. Furthermore, the invariants hold true for arbitrary interleavings of arbitrary protocol sessions.

Secure Coding Discipline. We emphasize that F*'s typecheck guarantees that we never violate any of the invariants sketched above. If the model would contain any flaws, for example, missing checks, incorrect handling of keys, incorrect modification of the state, etc., the corresponding function would fail this typecheck.

As mentioned in Section 6.3, we have implemented a faulty server that incorrectly iterates over domains during ownership verification, similar to the bug in Boulder [35] (see also Section 5). As expected, the typecheck fails, and we have to manually override F*'s typechecker to compile this code.[13]

---

[7]See the module ACME.Server.ReceiveChallengeVerification.

[8]See valid_acme_server_st in Application.Predicates.Helpers.

[9]See again the module ACME.Server.ReceiveChallengeVerification.

[10]See again valid_acme_server_st in Application.Predicates.Helpers.

[11]See the module ACME.Server.FinalizeOrder.

[12]See the module ACME.SecurityProperties.

[13]See the faulty implementation check_valid_and_set_ready_for_order_faulty in the module ACME.Server.Helperfunctions of our supplementary material [10].

7.2 Integrity Properties

We consider two main integrity properties: Integrity of an ACME flow from the server's perspective and the client's perspective. In the following, we present both properties and refer to Appendix B for the (simplified) DY* lemmas and the high-level proof structures.

Server-Side Integrity. The server-side integrity property states that any certificate that the server has issued for an honest client has actually been requested by the client, and the client owns the corresponding account key used for the protocol flow.

THEOREM 2 (SERVER-SIDE INTEGRITY). *For all possible global traces t of the DY* model of ACME, with s being a principal (an ACME server), d being a domain, D being a set of domains, acc_pub_key being some public key, and cert some certificate issued by s in t for the set D, if d is a domain in D, acc_pub_key is the account public key used for issuing cert, and the principal owning the domain d is honest, then the owner of d created an ACME order for the domains D and owns the private key to acc_pub_key.*

Client-Side Integrity. The client-side integrity property states that whenever the client stores a certificate (issued by an arbitrary, potentially dishonest server), then this certificate has been issued for the same set of domains that the client requested earlier.

THEOREM 3 (CLIENT-SIDE INTEGRITY). *For all possible global traces t of the DY* model of ACME, with c being a principal (an ACME client), D being a set of domains, and cert some certificate issued for the set D, if c stores the certificate cert in its state, then c previously created an ACME order object for the domains D.*

Similarly to the secure binding property, we specify both properties as lemmas in DY* and define suitable predicates, including those for valid traces, and prove these properties.

7.3 Proof Effort

For proving the security properties presented in the previous sections, we added 5,191 lines of proof-related code, which roughly corresponds to the number of functional lines of code of the ACME model (see Table 1). This number includes the proofs that follow from the trace invariants and all local proofs (see also Section 7.1, where we describe how local proofs are used to prove the secure binding theorem). Overall, the verification of the complete ACME model (including the typecheck of the DY* model) takes 67 minutes using an off-the-shelf machine. We highlight again that the verification is done modularly, and on average, each module is verified in less than 1.5 minutes. We estimate that the ACME analysis (creating the ACME model and proving the security properties) took several person-months, a large portion of which was for reading and transcribing the standard. We refer the reader to Section 9 for a comparison of our analysis to prior ACME analyses.

8 INTEROPERABLE IMPLEMENTATION

In the symbolic execution tests as described in Section 6.3, we confirmed our model to be internally consistent, i.e., our modeled ACME client and server are able to finish a protocol run when communicating to each other. While this provides some confidence in the correctness of our model within itself, we could still have modeled the ACME specification slightly incorrectly. So in a second step, we use our approach for interoperability (see Section 3.3) and augment our model with an (unverified) wrapper library to be able to connect our implementation to the real world. Here, we illustrate that our ACME client is indeed interoperable with a real-world server. To this end, we have our client run the ACME protocol with several different ACME servers, including *pebble* [44], an ACME server developed specifically for client compliance testing, as well as Let's Encrypt's staging and production servers based on *Boulder* [42], and show that all of them indeed issue certificates to our client.

As presented in Section 3.3, this wrapper library is written in OCaml and uses the ACME client APIs outlined in Section 6.2. On a technical level, the ACME model and the underlying DY* framework are compiled from F* to an OCaml library, against which we link our wrapper library. As with the ACME model itself, we provide the wrapper library and necessary configuration in our supplementary material [10].

Running the resulting interoperable client implementation with real-world ACME servers allowed us to find a bug in our model that did not surface in the symbolic execution tests from Section 6.3. As presented in Section 6.1, an ACME order object (in our model as well as in the specification) contains several sequences, e.g., a sequence of authorization URLs associated with the respective order object. The ACME specification does not require ACME servers to retain the order of elements in these sequences when including them in a response. Our initial client model, however, implicitly assumed to receive these sequences exactly as sent in the request. This bug did not surface in the symbolic execution tests, as our server model just reflects them in responses. But the *pebble* ACME (compliance testing) server [44] permutates them on purpose and thus leads us to the error. In addition to being quite subtle, this modeling bug only shows when ordering a certificate for more than one domain. Therefore, we emphasize the value of an executable model and using that ability to build confidence in the model's correctness.

9 RELATED WORK

As mentioned before, early drafts of ACME have been analyzed before. We will discuss these works as well as other work that focuses on public key infrastructures and the verification of crypto protocols in general.

Prior Analyses of ACME. Two prior works present formal analysis results for early drafts of the ACME standard. Bhargavan et al. [15] model domain authentication protocols of ACME draft 00 in ProVerif and analyze it for an authentication property similar to secure binding, in a symbolic cryptographic model without key substitution attacks. They introduce a notion of *strong identifier channels* for HTTP and DNS authentication (corresponding to Authenticated Send in Section 3.1) and show these mechanisms are strictly stronger than Email-based authentication (which is not supported anymore in the current standard). Jackson et al. [36] analyze the DNS-based domain authentication protocol in ACME drafts 00 and 02 in Tamarin. They show attacks on draft 00 if the signature scheme is vulnerable to key substitution attacks. Our treatment of no-CEO and no-DEO (see Section 3.2) key substitution is inspired by their work.

Both these prior works only model a small cryptographic core of the ACME protocol and focus on a narrow class of attacks, ignoring important RFC details like message formats and state management. The Tamarin model of Jackson et al. has ~100 lines of code, while the one of Bhargavan et al. has ~280 lines.

In contrast, we build the first in-depth model for the published ACME standard [5]. We account for certificates that cover multiple domains (which requires reasoning on unbounded data structures), while prior analyses only model a single domain. We also model JWS signatures and long-running sessions where domain lists are iteratively and asynchronously processed for domain verification, while prior analyses do not. Our specification is detailed enough to be seen as a reference implementation; it is both symbolically and concretely executable, and our ACME client interoperates with other ACME servers. Our security proofs account both for the high-level cryptographic attacks considered in prior work as well as low-level flaws in the message processing algorithms (see Section 5). This level of precision and rigor is reflected in the size of our verified ACME implementation (cf. Table 1).

Analyses of Public Key Infrastructures (PKI). Several prior works present novel PKI designs with security analyses in a variety of models. We refer the reader to [38] for a formal comparison between some of these proposals. ARPKI [7] and DTKI [56] seek to reduce the trust in certification authorities by using multiple CA servers and a public log. LocalPKI [27] is an IoT-friendly PKI design that allows local authorities to validate users and issues certificates. All three of these protocols were analyzed using the Tamarin prover, but none of the analyses focused on domain authentication, which is the main goal of ACME.

Verification of Crypto Protocol Standards. In recent years, symbolic and computational verification tools have been used to verify multiple real-world protocols (see [4] for a survey of the state-of-the-art). For example, the TLS 1.3 protocol standard was comprehensively analyzed using Tamarin, ProVerif, CryptoVerif, and F* [14, 22, 24]. The OAuth and OpenID standards were verified using a symbolic pen-and-paper model [28, 30, 31] as well as Tamarin [34]. This paper adds ACME to the list of formally analyzed protocol standards and closes an important gap in the security analysis of the web PKI, which is implicitly relied on by protocols like TLS.

One of the by far largest mechanized formal verification efforts of cryptographic protocols to date is the analysis of the 3,600 line miTLS implementation, which required 2,050 lines of type annotations for F7, as well as a large cryptographic proof that includes both manual arguments and a 3,000 line EasyCrypt proof [18]. In comparison, our analysis of ACME comprises more than 10,000 lines of functional and proof code, not counting the code of the DY* framework itself (see Table 1), again indicating the exceptional level of detail of our analysis and the scalability of our approach compared to other works in the domain of protocol analysis.

10 CONCLUSION

In this work, we have verified the security of the ACME standard [5] in an unprecedented level of detail. Our executable model of ACME in F* carefully accounts for all high-level protocol flows, including arbitrary recursive interleavings of multiple asynchronous sub-protocols between any number of clients and servers, and certificate

issuance for an arbitrary list of domains. In addition, it also covers important low-level protocol details, such as unbounded data structures and precise state management. Hence, our comprehensive model serves as a formal companion to the ACME RFC, and can be a useful guide for ACME implementors.

Our approach builds upon the DY* framework, which allows us to modularly specify and verify symbolic security properties for our ACME model using the F* programming language and type system. We extended DY* to support authenticated channels and key substitution attacks on signatures and use the resulting model to establish security theorems for ACME. These generic extensions are of independent interest and will also be of use in future cryptographic protocol verification projects using DY*.

Furthermore, we propose a general approach to make models written in DY* interoperable with real-world implementation to build further confidence that these models correctly reflect the protocol specification. Using this approach, we turn our model of ACME into an interoperable reference implementation and show that our ACME client can run the protocol with several real-world ACME servers, including the *Let's Encrypt* production server. Our interoperability approach relies on an unverified wrapper that connects our verified model with system libraries for networking and cryptography. As future work, we intend to generalize and verify this wrapper and parts of the underlying libraries, so that we can convert DY* models into high-assurance protocol implementations.

REFERENCES

[1] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth D. Schoen, and Brad Warren. 2019. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, New York, NY, USA, 2473–2487. https://doi.org/10.1145/3319535.3363192

[2] Andrew Ayer. 2015. ACME signature misuse vulnerability in draft-barnes-acme-04. https://mailarchive.ietf.org/arch/msg/acme/F71iz6qq1o_QPVhJCV4dqWf-4Yc/

[3] Michael Backes, Catalin Hritcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *J. Comput. Secur.* 22, 2 (2014), 301–353. https://doi.org/10.3233/JCS-130493

[4] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. 2021. SoK: Computer-Aided Cryptography. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, NY, USA, 123–141. https://doi.org/10.1109/SP40001.2021.00008

[5] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. 2019. Automatic Certificate Management Environment (ACME). RFC 8555. https://doi.org/10.17487/RFC8555

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2535838.2535847

[7] David A. Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2018. Design, Analysis, and Implementation of ARPKI: An Attack-Resilient Public-Key Infrastructure. *IEEE Trans. Dependable Secur. Comput.* 15, 3 (2018), 393–408. https://doi.org/10.1109/TDSC.2016.2601610

[8] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (2011), 8:1–8:45. https://doi.org/10.1145/1890028.1890031

[9] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, NY, USA, 535–552. https://doi.org/10.1109/SP.2015.39

[10] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY★ ACME Code Repository. https://github.com/reprosec/acme-case-study

[11] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY★: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, NY, USA, 523–542.

[12] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY★ Code Repository. https://github.com/reprosec/dolev-yao-star

[13] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. A Tutorial-Style Introduction to DY★. In *Protocols, Logic, and Strands: Essays Dedicated to Joshua Guttman on the Occasion of His 66.66 Birthday.* Springer. To appear.

[14] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, NY, USA, 483–502. https://doi.org/10.1109/SP.2017.26

[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Nadim Kobeissi. 2017. Formal Modeling and Verification for Domain Validation and ACME. In *Financial Cryptography and Data Security (Lecture Notes in Computer Science, Vol. 10322)*. Springer, Berlin, 561–578. https://doi.org/10.1007/978-3-319-70972-7_32

[16] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 2010. Modular verification of security protocol code by typing. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, New York, NY, USA, 445–456. https://doi.org/10.1145/1706299.1706350

[17] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with Verified Cryptographic Security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, NY, USA, 445–459. https://doi.org/10.1109/SP.2013.37

[18] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. 2014. Proving the TLS Handshake Secure (As It Is). In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8617)*. Springer, Berlin, 235–255. https://doi.org/10.1007/978-3-662-44381-1_14

[19] Simon Blake-Wilson and Alfred Menezes. 1999. Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol. In *Public Key Cryptography, Second International Workshop on Practice and Theory in Public Key Cryptography, PKC '99, Kamakura, Japan, March 1-3, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1560)*. Springer, Berlin, 154–170. https://doi.org/10.1007/3-540-49162-7_12

[20] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Found. Trends Priv. Secur.* 1, 1-2 (2016), 1–135. https://doi.org/10.1561/3300000004

[21] Buypass. 2020. Buypass Go SSL - Technical information. https://www.buypass.com/ssl/resources/go-ssl-technical-specification

[22] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, New York, NY, USA, 1773–1788. https://doi.org/10.1145/3133956.3134063

[23] Daniel McCarney. 2017. Validating challenges from multiple network vantage points. https://community.letsencrypt.org/t/validating-challenges-from-multiple-network-vantage-points/40955

[24] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, NY, USA, 463–482. https://doi.org/10.1109/SP.2017.58

[25] DigiCert. 2020. DigiCert - Certification Management. https://www.digicert.com/certificate-management/

[26] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983), 198–207. https://doi.org/10.1109/TIT.1983.1056650

[27] Jean-Guillaume Dumas, Pascal Lafourcade, Francis Melemedjian, Jean-Baptiste Orfila, and Pascal Thoniel. 2017. LocalPKI: An Interoperable and IoT Friendly PKI. In *E-Business and Telecommunications - 14th International Joint Conference, ICETE 2017, Madrid, Spain, July 24-26, 2017, Revised Selected Paper (Communications in Computer and Information Science, Vol. 990)*. Springer, Berlin, 224–252. https://doi.org/10.1007/978-3-030-11039-0_11

[28] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. 2019. An Extensive Formal Security Analysis of the OpenID Financial-Grade API. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE Computer Society, NY, USA, 453–471. https://doi.org/10.1109/SP.2019.00067

[29] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, NY, USA, 673–688. https://doi.org/10.1109/SP.2014.49

[30] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, New York, NY, USA, 1204–1215. https://doi.org/10.1145/2976749.2978385

[31] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, NY, USA, 189–202. https://doi.org/10.1109/CSF.2017.20

[32] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. 2011. Modular Code-Based Cryptographic Verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. ACM, New York, NY, USA, 341–350. https://doi.org/10.1145/2046707.2046746

[33] GlobalSign. 2021. GlobalSign - Auto Enrollment Gateway. https://www.globalsign.com/en/auto-enrollment-gateway

[34] Sven Hammann, Ralf Sasse, and David A. Basin. 2020. Privacy-Preserving OpenID Connect. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. ACM, 277–289. https://doi.org/10.1145/3320269.3384724

[35] Jacob Hoffman-Andrews. 2020. 2020.02.29 CAA Rechecking Bug. https://community.letsencrypt.org/t/2020-02-29-caa-rechecking-bug/114591

[36] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. 2019. Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, New York, NY, USA, 2165–2180. https://doi.org/10.1145/3319535.3339813

[37] Mike Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Signature (JWS). RFC 7515. https://doi.org/10.17487/RFC7515

[38] Hemi Leibowitz, Amir Herzberg, and Ewa Syta. 2019. Provable Secure PKI Schemes. Cryptology ePrint Archive, Report 2019/807. https://eprint.iacr.org/2019/807.

[39] Let's Encrypt. 2018. ACME v2 Production Environment & Wildcards. https://community.letsencrypt.org/t/acme-v2-production-environment-wildcards/55578

[40] Let's Encrypt. 2020. Let's Encrypt Has Issued a Billion Certificates. https://letsencrypt.org/2020/02/27/one-billion-certs.html

[41] Let's Encrypt. 2021. ACME Client Implementations. https://letsencrypt.org/docs/client-options

[42] Let's Encrypt. 2021. Boulder: An ACME-based certificate authority, written in Go. https://github.com/letsencrypt/boulder

[43] Let's Encrypt. 2021. Let's Encrypt certification authority. https://letsencrypt.org

[44] Let's Encrypt. 2021. Pebble ACME Server. https://github.com/letsencrypt/pebble

[45] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*. Springer, Berlin, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[46] Alfred Menezes and Nigel P. Smart. 2004. Security of Signature Schemes in a Multi-User Setting. *Des. Codes Cryptogr.* 33, 3 (2004), 261–274. https://doi.org/10.1023/B:DESI.0000036250.18062.3f

[47] Roger M. Needham and Michael D. Schroeder. 1978. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM* 21, 12 (1978), 993–999. https://doi.org/10.1145/359657.359659

[48] Thomas Pornin and Julien P. Stern. 2005. Digital Signatures Do Not Guarantee Exclusive Ownership. In *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3531)*. Springer, Berlin, 138–150.

[49] REPROSEC. 2021. REPROSEC Project. https://reprosec.org/

[50] Roland Bracewell Shoemaker. 2020. Automated Certificate Management Environment (ACME) TLS Application-Layer Protocol Negotiation (ALPN) Challenge Extension. RFC 8737. https://doi.org/10.17487/RFC8737

[51] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. 2018. All your clouds are belong to us: security analysis of cloud management interfaces. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2046660.2046664

[52] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. 2012. On Breaking SAML: Be Whoever You Want to Be. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. USENIX Association, Berkeley, CA, USA, 397–412.

[53] 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. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

[54] Mathy Vanhoef and Frank Piessens. 2017. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, New York, NY, USA, 1313–1328. https://doi.org/10.1145/3133956.3134027

[55] Thomas YC Woo and Simon S Lam. 1992. Authentication for distributed systems. *Computer* 25, 1 (1992), 39–52.

[56] Jiangshan Yu, Vincent Cheval, and Mark Ryan. 2016. DTKI: A New Formalized PKI with Verifiable Trusted Parties. *Comput. J.* 59, 11 (2016), 1695–1713. https://doi.org/10.1093/comjnl/bxw039

[57] ZeroSSL. 2021. ACME Automation. https://zerossl.com/features/acme/

A APPLICATION-SPECIFIC DATA STRUCTURES

As explained in Section 6.1, we did not just use "plain" symbolic bytes (as defined by DY*) to model ACME. Instead, we defined high-level data structures which closely resemble their respective definition in the ACME specification [5]. While this incurs some major benefits, as explained in Section 6.1, we have to convert these data structures to DY* bytes, since all interfaces to DY* are, of course, independent of ACME and thus require symbolic bytes. Note that this conversion is more than just a technicality of the model: If the conversion is wrong, we might lose all guarantees provided by DY*. In fact, we would not be able to prove that our modeled ACME client and server APIs do not violate our global trace invariants (see Section 7). Therefore, we not only have to implement suitable conversion functions, but we also have to prove that they are inverses of each other. In the following, we give an example of one of our data structures, the respective conversion functions, and correctness proof.

The following F* definition introduces a record type which models the inner JWS of an account key rollover message (see Section 7.3.5 of the ACME specification [5]), i.e., a message sent by a client who wants to update its account public key. As the name "inner" JWS suggests, an instance of this type is used as the payload of a regular ACME protocol message (i.e., a JWS signed with the current account key).

```
type acme_new_key_inner_jws = {
    alg: string; // Signature algorithm used to sign this JWS
    jwk: DY.Crypto.bytes; // The new (public) account key
    inner_url: url; // The same url as in the "outer" JWS
    inner_payload_account: url; // Account URL, a.k.a. kid
    inner_payload_old_key: DY.Crypto.bytes; // Old public key
    inner_signature: DY.Crypto.bytes // Signature over all of the above, using jwk
}
```

Given this definition, we implement a function to *serialize* an instance of acme_new_key_inner_jws to a single DY* bytes. Note that url is another of our high-level data structures for which we implemented similar serialization functions used here:

```
module DC = DY.Crypto // Define a shorthand

// Type of the serialization function
val serialize_acme_new_key_inner_jws: acme_new_key_inner_jws → DC.bytes

// Implementation
let serialize_acme_new_key_inner_jws input_object =
  DC.concat (DC.string_to_bytes input_object.alg) (
    DC.concat input_object.jwk (
      DC.concat (serialize_url input_object.inner_url) (
        DC.concat (serialize_url input_object.inner_payload_account) (
          DC.concat
            input_object.inner_payload_old_key
            input_object.inner_signature ))))
```

Of course, we also need a corresponding *parsing* function, turning DY* bytes into our high-level data structure. Note that the return type of this function is a result, i.e., can either be Success <value> or Error <error message>: If the bytes given to this function does not encode an acme_new_key_inner_jws, an Error is returned.

```
val parse_acme_new_key_inner_jws: DC.bytes → result acme_new_key_inner_jws

let parse_acme_new_key_inner_jws input_bytes =
  match DC.split input_bytes with
  | Error e → Error e
  | Success (ser_alg, t1) → (
    match DC.split t1 with
    | Error e → Error e
    | Success (jwk, t2) → (
      match DC.split t2 with
      | Error e → Error e
      | Success (ser_url, t3) → (
        match DC.split t3 with
        | Error e → Error e
        | Success (ser_acc, t4) → (
          match DC.split t4 with
          | Error e → Error e
          | Success (old_key, sig) → (
            match
              DC.bytes_to_string ser_alg,
              parse_url ser_url,
              parse_url ser_acc
            with
            | Success alg, Success url, Success acc → (
              let obj:acme_new_key_inner_jws = {
                alg = alg;
                jwk = jwk;
                inner_url = url;
                inner_payload_account = acc;
                inner_payload_old_key = old_key;
                inner_signature = sig
              } in
              Success obj
            )
            | _ → Error "Wrong␣format␣of␣acme_new_key_inner_jws" )))))
```

Finally, we need to prove that serialization and parsing are inverses of each other, i.e., we need to prove the following:

```
val parse_acme_new_key_inner_jws_lemma:
    inner_jws:acme_new_key_inner_jws →
  Lemma
    (ensures (
      let serialized = serialize_acme_new_key_inner_jws inner_jws in
      parse_acme_new_key_inner_jws serialized == Success inner_jws
    ))

val parse_acme_new_key_inner_jws_lemma2:
    t:DC.bytes →
  Lemma
    (requires (Success? (parse_acme_new_key_inner_jws t)))
    (ensures (
      let parsed = Success?.v (parse_acme_new_key_inner_jws t) in
      serialize_acme_new_key_inner_jws parsed == t
    ))
```

Thanks to F*'s theorem prover, we get the first lemma for free — we just have to ask F* to come up with a proof on its own:

```
let parse_acme_new_key_inner_jws_lemma obj = ()
```

To prove the second lemma, we have to provide some simple annotations to tell F* to use the corresponding lemma for values of the url type:

```
let parse_acme_new_key_inner_jws_lemma2 t =
  // Note that F* has to prove here that t is a concatenation of bytes. This is implied
  // by the precondition to this lemma (see its type above).
  let Success (ser_alg, t1) = DC.split t in
  let Success (jwk, t2) = DC.split t1 in
  let Success (ser_url, t3) = DC.split t2 in
  let Success (ser_acc, t4) = DC.split t3 in
  // Instantiate similar lemmas for the two bytes representing URLs
  parse_url_lemma2 ser_url;
  parse_url_lemma2 ser_acc
```

Similar functions and lemmas exist for all our high-level data structures.

B   DETAILS ON THE INTEGRITY PROPERTIES

In the following, we give more details on the integrity properties presented in Section 7.2, in particular, the (simplified) F* formulations for both properties and simplified high-level overviews of both proofs. We refer to our supplementary material [10] for the complete F* proofs.

B.1   Server-Side Integrity

The formulation of the server-side integrity property as an F* lemma looks as follows (slightly simplified for presentation):

```
val server_side_integrity_property:
  s:principal → // any (server) principal
  trace_idx:nat → // any trace index
  cert:certificate → // any certificate
  dom:domain → // any domain
  acc_pub_key:bytes → // any public key
  DY unit
    (requires (λ t0 → ( // any trace t0 ...
      valid_trace t0 ∧ // .. that is valid
      // the following predicate is true if the server s set its state at
      // trace index trace_idx with a state entry in which it stored the certificate
      // cert, the key acc_pub_key, and if the domain dom is one of the
      // domains for which the certificate is issued
      server_issued_certificate_for_domain_with_account_key s trace_idx
          cert dom acc_pub_key ∧
      // the following line requires that the principal that
      // owns the domain is not corrupted by the attacker
      ¬(is_principal_corrupted_before len(t0) (domain_principal_mapping dom)))))
    (ensures (λ t0 _t1 → (t0 == t1 ∧ (
        // the client that owns the domain ...
        let client = domain_principal_mapping dom in
        // sent an order request for the domains of the certificate ...
        client_sent_newOrder_request_for_domains client certificate.domains ∧
        // and owns the private key to the account public key
        // (ensured by the labeling of the key)
        client_owns_public_key client acc_pub_key
))))
```

Proof Structure. As already explained in Section 7.1, the state invariant of the server requires that whenever the server sets the status of an authorization to valid, the owner of the domain also owns the private key to the corresponding account public key contained in the verification response (recall that this guarantee is provided by the authenticated send predicate). The account public key sent in the verification response is stored by the server.

When receiving the CSR, the server verifies that the JWS containing the CSR is signed with this public key. Therefore, it follows

that the principal that owns the domain also owns the public key that the server associates with the certificate (i.e., the predicate client_owns_public_key holds true, see the post-condition of the F* representation of the theorem).

As already described in Section 7.1, the server obtains guarantees by the signature predicate when verifying the JWS. Besides the guarantees mentioned in Section 7.1 regarding the private key of the certificate, the signature predicate also states that the client that signed the JWS previously created an acme order with the exact set of domains as in the CSR (i.e., that the predicate client_sent_newOrder_request_for_domains is fulfilled).

These statements (essentially the post-condition of the theorem) are encoded in the state invariant of the ACME server.

As already described in Section 7.1, the final properties follow from the valid_trace predicate, which also contains the (application-specific) state invariants. Thus, the property can be easily proven from the guarantees provided by valid_trace.

B.2   Client-Side Integrity

The formulation of the client-side integrity property as an F* lemma looks as follows (again slightly simplified for presentation):

```
val client_side_integrity_property:
  c:principal → // any (client) principal
  trace_idx:nat → // any trace index
  cert:certificate → // any certificate
  DY unit
    (requires (λ t0 → ( // any trace t0 ...
      valid_trace t0 ∧ // .. that is valid
      // the following predicate is true if the client c set a state at trace index
      // trace_idx with a state entry in which it stored the certificate cert
      client_stores_certificate c cert trace_idx
      )))
    (ensures (λ t0 _t1 → (t0 == t1 ∧ (
        // the following predicate is true if the client c set a state prior to
        // the trace index trace_idx in which it stored an ACME order with the
        // domains of the certificate cert
        domains_previously_stored_in_order c cert.domains trace_idx
      ))))
```

Proof Structure. As for the previous properties, the client-side integrity property is essentially implied by the trace invariant. Again, the predicate that is central to this proof is the state invariant. The state invariant of the ACME client requires that whenever the client stores a certificate that it receives, there is a state entry in which it stores information about a CSR (previously created by the client). More precisely, the set of domains of the certificate must be the same as the domains used for this CSR.

As already explained previously, the client function that stores the certificate needs to ensure that this property holds true, or else, the client would not be able to update its state.

Furthermore, the client state invariant has several requirements on the state entries used for storing the CSR information mentioned above. One of these requirements is that the client previously updated its state, and in this state, stored a state entry with data contained in an ACME order. More precisely, this order needs to contain the same set of domains as stored for the CSR state entry.

Thus, when storing the CSR, the corresponding client function needs to ensure that this condition holds true. In this case, when constructing the CSR, the client uses the domains of an ACME order stored in one of its state entries, and therefore, it can easily be shown that the condition of the state invariant holds true.

Relevant sections of [5]	Coverage in our model (definitions in modules)
6. Message Transport	
6.1 HTTPS Requests	(We overapproximate by not using TLS, see modeling decisions in Section 6.)
6.2. Request Authentication	generate_signature_for_jws_acme_request (ACME.Data), verify_jws_acme_request (ACME.Server.HelperFunctions)
6.3. GET and POST-as-GET Requests	gen_http_request_with_server_domain_in_header (ACME.Client.HelperFunctions)
6.4. Request URL Integrity	generate_jws_acme_request (ACME.Data), verify_jws_acme_request (ACME.Server.HelperFunctions)
6.5. Replay Protection	acme_server_new_nonce (ACME.Server), generate_jws_acme_request (ACME.Data), client_finds_valid_replay_nonce (ACME.Client.HelperFunctions),
6.6. Rate Limits	(We do not enforce rate limits, a safe over-approximation.)
6.7. Errors	(We do not model aborting error messages. See also the discussion in Section 6.4.)
7. Certificate Management	
7.1. Resources	
7.1.1. Directory	(The directory only contains static meta-information about the server. We do not model the directory.)
7.1.2. Account Objects	Type acme_account (ACME.Data)
7.1.3. Order Objects	Type acme_order (ACME.Data)
7.1.4. Authorization Objects	Type acme_authorization (ACME.Data)
7.1.5. Challenge Objects	Type acme_challenge (ACME.Data)
7.1.6. Status Changes	Type acme_status (ACME.Data)
7.2. Getting a Nonce	acme_server_new_nonce (ACME.Server), acme_client_request_replay_nonce_nw (ACME.Client), acme_client_receives_and_saves_replay_nonce_nw (ACME.Client)
7.3. Account Management	We support account key rollover with the function acme_server_update_account_key (ACME.Server)
7.4. Applying for Certificate Issuance	Client's functions (ACME.Client): acme_client_orders_certificate_http, acme_client_sends_CSR_http, acme_client_retrieves_certificate_http, acme_client_receives_and_saves_certificate Server's functions (ACME.Server): acme_server_new_order, acme_server_finalize_order, acme_server_retrieve_cert
7.5. Identifier Authorization	Client's functions (ACME.Client): acme_client_send_authorization_request_i_http, acme_client_receive_authorization_response_http, acme_client_triggers_challenge_i_http, Server's functions (ACME.Server): acme_server_identifier_authz, acme_server_challenge_response
7.6. Certificate Revocation	(Certificates in our model are valid indefinitely and cannot be revoked. See also the discussion in Section 6.4.)
8. Identifier Validation Challenges	
8.1. Key Authorizations	acme_client_challenge_response_http (ACME.Client), acme_server_receive_challenge_verification_http (ACME.Server)
8.2. Retrying Challenges	(Challenges in our model remain in the *processing* state in case the validation does not succeed. See modeling decisions in Section 6.)
8.3. HTTP Challenge	acme_client_challenge_response_http (ACME.Client), acme_server_trigger_challenge_verification (ACME.Server), acme_server_receive_challenge_verification_http (ACME.Server)
8.4. DNS Challenge	(As discussed in Section 6.4, we only model the HTTP challenge, which is structurally similar.)

Table 2: Relevant sections of the ACME standard and their counterparts in our model

From these restrictions on the client state implied by the state invariant, it follows that whenever the client stores a certificate in its state, it previously must have created an ACME order with the same set of domains as in the certificate.

C   COVERAGE OF THE ACME STANDARD

We illustrate the coverage of our model w.r.t the ACME standard in Table 2. In this table, we list all relevant sections from [5] that describe either data structures or functionalities and point to places in our model where the respective part is implemented. See our supplementary material for the full code [10].