A Formal Model for Delegated Authorization of IoT Devices Using ACE-OAuth

LUCA ARNABOLDI, Newcastle University, UK
HANNES TSCHOFENIG, Arm Limited, UK

As our daily lives become ever more connected, the need for security becomes more important. This need is, however, in conflict with the way most Internet of Things (IoT) devices are designed. These devices often have limited computing power, memory, and security mechanisms to defend against a range of attacks. The protocol designer must therefore take all these constraints into consideration when making IoT security design decisions. To bring authentication and authorization solutions to constrained IoT devices, the Internet Engineering Task Force (IETF) Authentication and Authorization for Constrained Environments (ACE) working group[1] formed and profiled the OAuth protocol suite. Reusing already existing security protocols offers many benefits, since designing security protocols is a complicated task. Over the years, several technologies for automated verification, such as Tamarin [6] and Easycrypt [1], have been developed by researchers. Given a protocol design and a set of security properties (such as secrecy and authentication), these tools can help prove that these properties hold for any potential scenario, under a given threat model. These tools have become popular as researchers find prominent internet security protocols vulnerable. In this paper, we use the formal analysis tool Tamarin to analyze the ACE-OAuth protocol, and illustrate how protocol designers can analyze their own extensions.

Additional Key Words and Phrases: IETF, OAuth, ACE-OAuth, protocol verification, security, standardization

1 INTRODUCTION

IoT devices often have limited processing power, storage space, and transmission capacity. In many cases, these devices do not provide user interfaces, and are meant to interact without human intervention. There is a need to provide authentication and authorization capabilities for such devices when they are connected to the internet. The protocol used today to provide delegated access for web applications and services is the OAuth 2.0 protocol. With the standardization work in the IETF ACE working group, engineers have applied the OAuth 2.0 framework to IoT devices, called ACE-OAuth.

In a paper submitted to the third OAuth Security Workshop, we analyzed a small subset of the ACE-OAuth protocol (namely, the Client Token Protocol) using Scyther and Avispa, two tools for automated verification. Consequently, the Client Token Protocol was removed from the specification. In this paper, we go further and apply a more modern tool, namely Tamarin, to the entire ACE-OAuth framework.

The main contributions are:

- **Formalization of the ACE-OAuth architecture:** We extract security requirements and security goals from the specification and describe them formally. As explained in Section 4.1, this step is subject to interpretation. The properties were not specified formally in the ACE-OAuth specification [7] or any of the accompanying documents. It is uncommon to find any formally specified security goals in IETF specifications.
- **Formal model of a ACE-OAuth case study:** After formalizing the specification, we present a fully executable model of the ACE-OAuth protocol. To our knowledge, this is the first automatically analyzable, fully verifiable

[1]The IETF is a Standards Developing Organization (SDO). The technical work in the IETF happens in working groups, which are clustered into areas. There are typically more than 100 active working groups at any particular time.

Authors' addresses: Luca Arnaboldi, Newcastle University, 1 Urban Science Square, Newcastle University, NE4 5TG, UK, l.arnaboldi@ncl.ac.uk; Hannes Tschofenig, Arm Limited, 110 Fullbourne Rd. Cambridge, UK, hannes.tschofenig@arm.com.

model for this particular protocol. We modeled the protocol so we can perform a full security analysis, to work around standardized lemmas and requirements (for applicability and higher assurance of proof).

Analysis and Results: We carry out a full, formal security evaluation of the protocol as a whole, then investigate a particular implementation. As the spec allows for several deployment setups, we asked our colleagues to formulate a profile adhering to the ACE-OAuth implementation, and carried out a comprehensive analysis.

We hope that by making the formal model of the ACE-OAuth framework available, it can be a baseline for analysis of future protocol extensions. As-is, it should be possible to test different ACE-OAuth setups by altering the current model, as exemplified by our case study. We intend that readers use the model to make informed decisions about protocol changes and extensions.

2 FORMAL PROTOCOL VERIFICATION OVERVIEW

With the increasing number of protocols purpose-built for IoT, it becomes more difficult to manually analyze security. Hence, we offload the analysis to automated verification tools. The way these protocols are formally verified can be split into two main methods: using a symbolic [3] or computational [8] model. These approaches have different objectives. The symbolic model is mainly used to verify protocol specifications, as it is very high level and quicker to implement. The computational model is more precise and can analyze specific implementation details, but is more time-consuming.

In the symbolic model, cryptographic primitives are represented by function symbols or black boxes. The messages are terms on these primitives, and the adversary is restricted to compute with a predefined set of strategies. This model assumes perfect cryptography, so there is no notion of brute forcing a password (unless specifically introduced). One can add equations to model algebraic properties of the cryptographic primitives. The only equalities that hold are those explicitly given by these equations. The main adversary in this model is described as the Dolev-Yao adversary [3], and the channel, or medium of transfer, is the intruder. Hence, anything sent on this channel can be read or modified. The adversary can fabricate and replay all messages sent over the channel. The limitation of this method is that the adversary cannot encrypt/decrypt messages without a key. This overly simplistic model can capture errors in design logic. However, it cannot fully describe situations where the cryptographic primitives cannot be treated as black boxes, or when security properties are specifically defined.

Conversely, in the computational model, the messages are bitstrings taking a specific form. The cryptographic primitives are functions from bitstrings to bitstrings. Unlike the Dolev-Yao model, the adversary is any probabilistic Turing machine. A proof passes if the adversary can only compute the key or break the requirement with negligible probability. The length of keys is determined by the value 'security parameter', and the run-time of the adversary is polynomial in the security parameter. A proof in this model is more rigorous and closely related to the eventual implementation. This gives us more assurance of the protocols security, which is especially true when outlining edge cases. For example, a newly designed protocol still supports older cryptographic algorithms for legacy reasons.

When designing a protocol, the symbolic model offers more flexibility as changes to the protocol design can be verified more quickly. We have selected Tamarin as our automatic verification tool, since it has been used successfully to analyze TLS 1.3 and several other prominet protocols.

2.1 Tamarin

The Tamarin prover is described as: "A powerful tool for the symbolic modeling and analysis of security protocols" [6]. Its input is a security protocol model, specifying the actions taken by agents running the protocol in different roles, such

as the protocol initiator, the responder, and the trusted key server, a specification of the adversary, and a specification of the protocol's desired properties. Tamarin provides general support for modeling and reasoning security protocols. Protocols and adversaries are specified using an expressive language based on multiset rewriting rules. These rules define a labeled transition system whose state consists of a symbolic representation of the adversary's knowledge, the messages on the network, information about freshly generated values, and the protocol's state. The adversary and protocol interact by updating network messages and generating new messages. Security properties are modeled as trace properties, checked against the traces of the transition system, or in terms of the observational equivalence of two transition systems. It is fully automated, but one can still use an interactive view to guide the proof search. If the tool's automated proof search terminates, it returns either a proof of correctness (for an unbounded number of role instances and fresh values) or a counterexample, representing an attack that violates the stated property.

To model a protocol, Tamarin uses a concept known as multiset rewriting rules. The rules operate on the system's state, which is expressed as a multiset (i.e. a bag) of facts. We explain Tamarin syntax using a toy example:

```
rule ExampleRule:
     [Fr(~UUID)]--[DeviceRecorded($X,~UUID)]->[!RegisterDevice($X,~UUID)]
```

Tamarin rules consist of a left-hand side, actions and a right-hand side. Within these rules, the basic ingredients are:

Terms: Terms are variables in the system of the types ~x denotes ~x:fresh (like a nonce), $x denotes x:pub (publicly available value such as a device IP), and m denotes m:msg (untyped message).

Facts: Facts are ways that variables are derived, rules use facts to manipulate terms in the system.

Actions: Actions are unique and differ to the other two ingredients as they do not alter the state of the protocol. They are used as annotations of what is happening in the rule and are used exclusively in the property specification.

A rule can only be executed if all the facts on its left-hand side are available in the current state. When a rule is executed, it will consume the facts on the left, removing them from the global state, and produce facts on the right, adding them to the global state. In the running example, we see fact `!RegisterDevice(C,~UUID)`. When called, this fact takes a public value 'C', representing the client, and relates it to a unique value representing the unique device ID. If the fact has "!" in front, it can be consumed multiple times and is persistent in the global state.

3 ACE-OAUTH OVERVIEW

Since the purpose of this paper is to formally analyze the ACE-OAuth protocol we need to briefly describe how it works first. The full details can be found in [7]. The high-level architecture of ACE-OAuth is shown in Figure 1 where the OAuth Client interacts with an Authorization Server to obtain an access token. Once the OAuth Client is authenticated and authorized (potentially with the consent of the resource owner or some third party), it receives an access token to use with a Resource Server. The Resource Server hosts the protected resource that the OAuth Client tries to access. Unlike a classical OAuth that uses bearer tokens, an access token used with ACE-OAuth is a proof-of-possession token, which has a key bound to it. When the OAuth Client presents the access token to the Resource Server, it also needs to demonstrate possession of the key associated with the access token.

ACE-OAuth was designed for use in IoT environments. As the name indicates, it uses the OAuth framework. In order to make OAuth suitable for IoT, more lightweight protocols may be necessary. For example, CoAP may be used instead of HTTP, DTLS instead of TLS, CBOR instead of JSON, and COSE instead of JOSE. These protocol and encoding alternatives ensure that the payloads transmitted over the air are significantly reduced in size. While many encoding

details are not necessarily relevant for a formal analysis, they are useful for understanding the goal of the standardization activity in the IETF ACE working group. There are, however, subtle implications of switching from one protocol to another, since protocols often have slightly different semantic and properties.

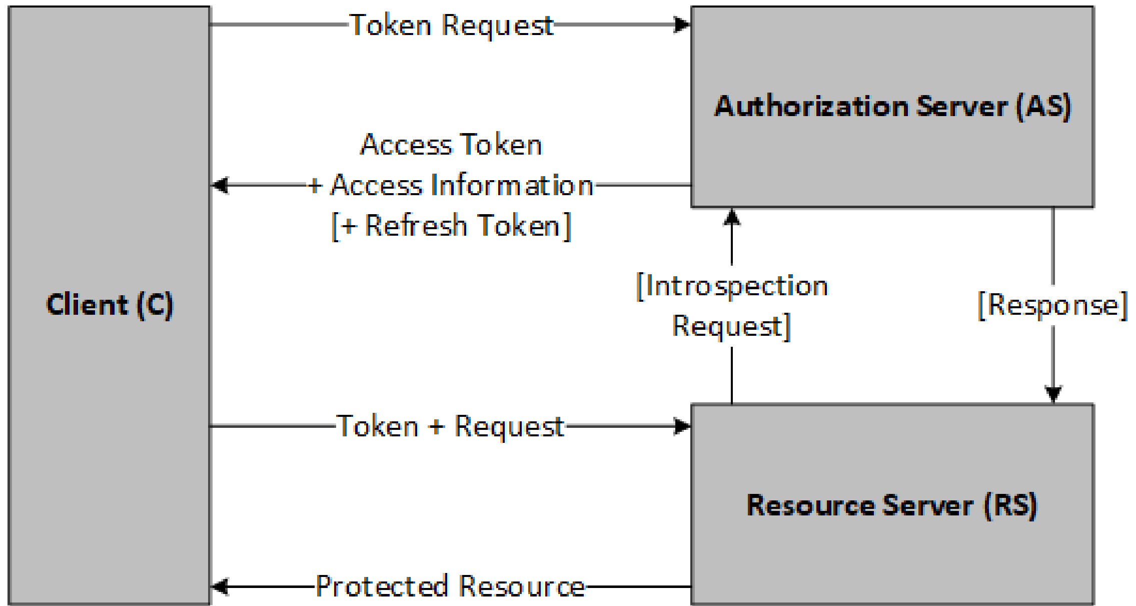

Fig. 1. Protocol Flow of ACE OAuth without implementation details or security properties

4 SECURITY ANALYSIS OF ACE-OAUTH

In this work, we focus on a security analysis of the protocol flow as described in the specification. There is a careful level of abstraction we must choose to follow as a consequence. IETF specifications are often generic by design and offer a range of options. We want to maintain a high a level of granularity to ensure accuracy and fealty to the original specification.

4.1 Security Requirements

In order for an authentication protocol between two parties to be considered as functioning correctly, we must first discuss the meaning of a successful authentication protocol. We consider protocols that aim to authenticate two IoT devices, referred to as agents, with or without help of a third party (such as an authentication server). We use the word "role" to refer to the purpose an agent takes in the protocol run (i.e. initiator, responder, or server). So to check whether the protocol successfully achieved this, we must, for every run of the protocol, check that this definition holds. As described in Lowe and Gavin's seminal work [4], the following properties must hold in order to successfully deploy an authentication protocol, starting from the weakest definition of aliveness to the strongest of injective agreement. We also refer to these conditions as security properties, as if these properties hold, it means that the authentication is successful and, the attacker cannot break the protocol or impersonate one of its participants.

The properties are:

Goal 1 - Aliveness: A protocol guarantees to an agent *a* in role A aliveness of another agent *b* if, whenever agent *a* completes a run of the protocol, apparently with agent *b* in role B, then agent *b* has previously been running the protocol.

Goal 2 - Weak Agreement: A protocol guarantees to an agent *a* in role A a weak agreement with another agent *b* in role B if, whenever agent *a* completes a run of the protocol, apparently with role b, then agent *b* has previously been running the protocol with A.

Goal 3 - Non-Injective Agreement: A protocol guarantees to an agent *a* in role A a non-injective agreement with another agent *b* in role B on a message t if, whenever an agent *a* completes a run of the protocol, apparently with role B, then agent *b* has previously been running the protocol with agent *a* and they both agreed on the message t.

Goal 4 - Injective Agreement: A protocol guarantees to an agent *a* in role A a non-injective agreement with another agent *b* in role B on a message t if, whenever an agent *a* completes a run of the protocol, apparently with role B, then agent *b* has previously been running the protocol with agent *a* and they both agreed on the message t. Additionally, there is a unique matching partner instance for each completed run of an agent, i.e. for each Commit by an agent there is a unique Running by the supposed partner (see Protocol in Appendix A for details).

More protocol-specific properties can be easily added without changing the formal specification of the protocol. The added properties verify that the flow of the protocol is not altered, and it adheres to the original protocol. The properties can aid to check that the addition of changes does not infringe on the security requirement and that they do not introduce vulnerabilities.

4.2 Formalizing the ACE-OAuth Security Requirements

In an effort to ensure correctness, we analyzed the specification and translated each requirement into formal definitions of security, as well as determining what level of security is needed at each level (and the downsides if the level is not met). It is worth noting that as the standard allows for different implementations, some security specifics are implicit and not covered. For example, there is an assumption that state-of-the-art cryptography is used. Hence, we do not investigate brute force attacks against keys.

We break the protocol down into three phases and formalize the requirements of each phase:

(1) The first phase is the token grant phase. A device makes a token request to the Authorization Server and specifies the need for the token. Upon being verified, a token is granted.
(2) The second phase is the resource request phase. Upon having been granted a token, the device proceeds to use the token to access a resource. The resource server learns about the granted permissions through the token.
(3) The third and final phase is the token authorization and proof-of-possession phase. The protocol specification allows the token to be verified locally at the resource server, or for the use of the token introspection mechanism. Once this step is successfully completed, the device is granted access.

The requirements for the secure and correct functioning of the protocol as per the specification are discussed and formalized below (in the format <Document Name> <Section> <Page Number> <Requirement Number>). We explain modeling choices, how to use the model for any given scenario, and to test variations.

4.3 Token Grant

The token grant is the core principle behind OAuth, and within it lies the essence of the protocol. This enables for an untrusted user to authenticate with a potential resource and gain access to its content. Consequently, if the token is unprotected, an attacker can arbitrarily access resources in the system, so it is the single point that must be secured most efficiently.

> **ACE-OAuth, Section 6, p37, r1:** The AS must provide confidentiality protection for any interaction with the client.

When we talk about confidentiality of information, we are talking about protecting the information from disclosure to unauthorized parties. Whilst this is often associated with encryption, the protocol leaves this up to interpretation. We choose to keep to this interpretation, but it could be possible to ensure this property in other ways. Formally, we require that for all messages in between the AS and C, an Adversary may not know the content of the message unless the private encryption key of either party has been exposed.

As this is an authentication protocol, there is an expectation that the AS and client are able to mutually authenticate each other. There is an underlying assumption that the client has previously registered with the specific AS.

> **ACE-OAuth, Section 5, p15, r2:** It is REQUIRED that the endpoint and the AS are mutually authenticated and the responses MUST be bound to the request.

Since we assume that the device has been pre-registered with the AS, there is some means for the client to know that the client is legitimate. The AS must therefore demonstrate injective agreement on the request to the client C. This is a strong assumption; perhaps there are some environments in which this might not be needed (as we will discuss in Section 5.4). However, since the intent is to model the spec accurately for future use, we keep these requirements strict. If one wishes to test different setups to ensure the requirement holds, this would require a switch from the modeled secure channel to an authenticated one (provided in the model) with their desired cryptographic scheme. The requirement should then be re-checked and the setup can be altered until it is successfully met.

4.4 Resource Request

Unlike the Token Grant phase where a client is pre-registered with the AS, the initial communication between C and the resource server (RS) is insecure. The client must provide proof that he has access to the resource, and that the client is the intended owner of the token.

> **ACE-OAuth, Section 5.1.1, p16, r2:** A resource request MUST be rejected if: the request is sent on an unprotected channel, the access token is invalid or not present, or the token is valid but not for the specified resource.

These requirements might seem reasonable, but their ambiguity leads to potential issues in the protocol setup. If there is no previous registration between the C and the RS, an unprotected channel is the norm. An unprotected channel, cryptographically speaking, is one to which an adversary has read, write, and intercept access.

We proceed by modeling this as a non-protected channel, as there is no means for previous authentication. Authentication must be performed afterwards to ensure that the requirements are met.

> **ACE-OAuth, Section 5.8.2, p35, r1:** If a client makes a request to the RS, it MUST show that is has the access token that validates this request.

As the token is bound to a specific party, leakage of the token on an insecure channel does not imply that an adversary is able to access the resource. However, if there is no second authentication phase, the request should not be accepted.

> **ACE-OAuth, Section 6, p39, r1:** The secure channel between the client and RS must provide encryption, integrity, and replay protection.

The requirement here is not specific on how to achieve this, and can be met in multiple ways. Formally, the RS must provide injective agreement to the client on the request. Subsequent granting of the requested resource must not be known to an attacker unless the private encryption key of either party has been leaked.

For the purpose of usability, we formally verify that if these conditions are met, the desired properties hold and the protocol can be securely executed to completion. In our case study, we investigate one method in which this is ensured through use of Signing and Asymmetric encryption on the request and token. Once again, however, multiple scenarios exist, including the addition of the further phase of token introspection.

4.5 Token Introspection

Though this step is optional in the ACE-OAuth specification [7], it represents one of the most useful features of the framework. In opposition to pure OAuth, the AS in ACE may aid in authenticating the client on behalf of the RS, as well as create a secure channel between them. As such, it is essential that the communication is properly secured. This is specifically pertinent to some low-power scenarios that may struggle to perform complicated asymmetric exchanges to ensure authentication.

> **ACE-OAuth, Section 5.7, p27, r1:** The communication between the AS and RS must be integrity protected and encrypted.

> **ACE-OAuth, Section 5.7, p27, r2:** The communication messages between the AS and the RS must have a binding between request and response.

> **ACE-OAuth, Section 5.7, p27, r3:** The AS must ensure the requesting entity has the right to request information about the specific token.

These requirements match the previously formalized requirements and hence, we do not repeat them. What we want to investigate is whether security is achievable with these requirements in place, permitting safe protocol flow. We also want to investigate whether they apply to all possible variations of protocol setup. We asked ourselves the following questions: Given a set of requirements how can we test that they are all needed? If different setups are tested, do the same requirements apply?

We attempt to answer these questions in the case study, and demonstrate how to answer them with use of automated verification tools.

5 EXAMPLE EXTENSION: SMART FACTORY

We asked our co-workers to design a profile of the ACE-OAuth framework based on a use case important to them. Developing a profile is necessary to restrict the number of options available within the ACE-OAuth framework. We also

wanted to explore the validity of our model and determine whether our security requirements were complete. Finally, we wanted to know how easy it would be to apply the model to an extension. During these discussions, we noticed that the translation of the extension into a formal model helped spur useful questions.

The purpose of the extension was to use the ACE-OAuth framework in a smart factory where technicians make changes to machines on the factory floor, as well as perform firmware updates. Only authorized technicians working for the company should be allowed to run maintenance jobs on machines. Hence, there is a need for the technician to authenticate with the AS that issues the tokens.

Only public key cryptography was used in the profile. The client, which runs on a tablet, is provisioned with a trust anchor of the AS and has a certificate plus a private key. The AS is provisioned with a certificate and private key, and the RS is also provisioned with the trust anchor of the AS, and a certificate plus the corresponding private key.

After the Client authenticates the AS (and vice versa), the AS creates a token. This token is digitally signed by the AS and includes the proof-of-possession key (i.e. a public key provided by the Client with the token request).

When the Client interacts with the RS, it not only needs to make a resource request and include the token, but also demonstrate possession of the private key that corresponds to the public key included in the token. Since communication between the Client and RS in this profile uses Bluetooth Low Energy, a new payload was defined that encapsulates the resource request, timestamp, and token. We called this payload the 'Operation Bundle". This Operation Bundle is digitally signed with the private proof-of-possession key.

The RS did not use token introspection in our profile, but processed the self-contained token locally. This required the RS to:

- Use the stored trust anchor to verify the digital signature covering the token.
- Extract the proof of possession public key from the token.
- Use the public key to verify the digital signature covering the Operation Bundle.
- Check for expiration.
- Determine whether the Client was allowed to access the protected resource based on what was contained in the claims of the tokens.

5.1 Security Properties of the Smart Factory Use Case

As highlighted in the previous section, ACE-OAuth allows for flexible deployment. To make a detailed security analysis, we must choose a subset of the offered features that are useful in a given scenario.

What we wanted to demonstrate with this case study is that it is possible to find security implications of each design decision allowed by the specification. This allows us to evaluate the security implications of each decision, and generate a set of minimum requirements for different protocol variants. Depending on core security requirements in the overall protocol, it may sometimes not be possible to have that level of assurance, given specific implementation details.

Protocol verification is not black and white, but rather allows for considerable freedom to work toward a specific scenario that matches one's needs, while still giving assurance that the analysis is correct for the specific setup. Below, we show an example of how this is done, starting from: the initial analysis, finding minimum security requirements, and risk assessment in specific scenarios.

5.2 Token Grant

The ACE-OAuth specification determines that there should be a secure channel between the Client and AS. There is, however, no mention of requiring a secure channel between the user (in this case, a technician) and the AS. While this is out of scope of the ACE-OAuth protocol itself, it is not an unlikely scenario, given the specific environments that ACE-OAuth is aimed toward. Of course, since OAuth also considers out-of-scope user authentication, it was logical to follow the same design spirit also in ACE-OAuth. It is, however, worthwhile to think about the implications of this user authentication step, and how to model it.

5.3 Resource Request & Confirmation - Operation Bundle

The Operation Bundle is a construct that cannot be found in the ACE-OAuth specification itself but would, at least conceptually, be present in one of the ACE-OAuth profiles explaining the interaction between client and RS. As previously described, the idea of the Operation Bundle was to combine the resource request, the token, and a timestamp together, and subsequently protect this bundle with a digital signature. The Operation Bundle corresponds to an application layer remote procedure call that uses CBOR/COSE.

5.4 Discussion

Automated verification tools are often thought of as ways to “break protocols". However, the main reason to use these tools is to ensure protocol security. Designing a verification tool is much harder, because to have assurance of security, one must cover every scenario. Finding an attack only requires a single insecure scenario to be found, as argued in [2, 5].

As a limitation, a protocol designer has to find the appropriate level of abstraction when translating a specification into formal language, and correctly implement the protocol in Tamarin.

The initial phase was to test out a specific case study and gather the full results in Section 5.4, then use the results to generate new requirements to ensure the security of the extension. Then, we discussed these in terms of the protocol to show how the written spec can be extended and adjusted to fit our analysis.

We kept strictly to the specification requirements, in two phases:

(1) Token grant requirements: secrecy of request, injective agreement on the request, and injective agreement on the token.
(2) Resource access: secrecy of resource, and injective agreement on the resource.

The results of a naive implementation of the ACE-OAuth profile are shown in Table 1, which lists several vulnerabilities. We will fix these vulnerabilities, since they are in part introduced by the flexibility of the ACE-OAuth framework. While this initial analysis might seem alarming, these results are caused by small and easy-to-fix design decisions.

		Secrecy	Injective-Agreement
Token Grant	Token	✓	X
	Request	✓	X
Resource Request	Response	X	X

Table 1. Naive Implementation of ACE-OAuth Profile

5.5 Generation of Minimal Requirements

By generating minimal requirements, we wish to enable the implements of ACE-OAuth to have guided means to test variations for ACE-OAuth. One of the key characteristics of IoT is its dynamicity and heterogeneity, which allow it to apply to vastly different scenarios. The protocol allows for several profiles to be constructed from the base protocol. However, each variation introduces subtle differences to the protocol that, in turn, could introduce a vulnerability. To avoid introducing vulnerabilities, a set of minimal requirements must be specified for each variation allowed. We define the minimal requirements as the smallest possible requirements in order to still meet the outlined security goals. These can vary depending on what rigidity of goal one has and as can be seen they can differ. One can therefore make exact decisions of how to design their protocol to meet the security objectives of their project. We outline our methodology for the above case-study, given the results shown in the Section 5.1, we adjusted for the possible attacks on the protocol we discovered and made adjustments the setup.

Sometimes it is not possible to adjust the setup in a real-world environment, in which case it is still worth knowing which threats the design decision leaves us open to. We discuss below why there is still a risk of token leakage in our current setup, as there is a vulnerability with the resource request found in the specification.

To generate the requirements, we introduced a bottom up approach, involving adding layers of security until the requirements were met. This investigation proved useful, as it uncovered design flaws, as well as variations in the security necessary to meet the goals. We highlight the set of security procedures needed to achieve the outlined goals in the tables below. We use minimal assumptions (additionally to the spec-required ones) for the token request to obtain the desired security properties. For our specific analysis, we assume that an attacker cannot access any information created by participants in the protocol until it is sent out on a communication channel.

		Secrecy	Injective-Agreement
Token Grant	Token	$\neg K$	$\neg K \wedge S_{CA}[*]$
	Request	$\neg K$	$\neg K$

Table 2. Updated Security Requirements for Token Grant of ACE-OAuth Profile

The table should be read as follows: We use notation K to represent key reveal so $\neg K$ means assumption holds if the key is secret, and notation $S_{CA}[*]$ to represent a secure channel between two parties (in this case, C and AS) with the variables that need to be passed on that channel in the [ ], and SR_{CA} confirming the need for a replay-protected secure channel.

What was of more interest was that the requirements could be loosened for part of the protocol and still achieve the same security. The timestamp is only used to avoid replay of the token, as the request in phase two cannot be replayed due to the request only matching the token. This also applies to the full ACE-OAuth spec, and is something that should be taken into consideration for potential extensions.

This allows us to loosen the requirement so a replay-protected channel is not necessary. This change still allows us to meet full security goals, while increasing performance and reducing strain on devices.

Table 3 uses the same notation as Table 2, and V indicates that signature verification is needed (in our case, through a public key infrastructure).

In phase two, three aspects need to be considered in modeling:

		Secrecy	Injective-Agreement
Resource Request	Response	$V[T, B, ts] \wedge SR_{CR}[*] \wedge \neg K[*]$	$V[T, B] \wedge SR_{CR}[*] \wedge S_{RC}[]$

Table 3. Updated Security Requirements for Resource Request of ACE-OAuth Profile

(1) Timestamps are used for replay protection in the Operation Bundle. The RS checks for replays using locally available time information.
(2) Token introspection is not used in our profile so the RS is not able to communicate with the AS in real time.
(3) The token itself is public information, and does not require confidentiality protection.

In some scenarios, a requirement may not be implementable due to device constraints. While not necessarily advisable, one may relax certain requirements while still following the protocol for interoperability. In such a case, it is important to determine what vulnerabilities are thereby introduced.

6 CONCLUSION

Analyzing security protocols using formal methods is a valuable way of demonstrating that the modeled protocol meets indicated security goals. It also helps justify design decisions. In this paper, we focused on ACE-OAuth and an extension we developed for illustration purposes. We believe using tools like Tamarin makes formal protocol verification easy enough for protocol designers to utilize. We present a methodology to automatically verify extensions to the core flow of ACE-OAuth using Tamarin, and show the validity of the approach through our case study. The model can guide designers and engineers alike in making safe and secure decisions. By editing small portions of the model, new avenues of attack are introduced and consequently found through formal analysis. We firmly believe that using formal models for protocol verification is no longer optional, and seek to provide a means to simplify the burden for future work on the ACE-OAuth protocol by using our model.

During our work we made the following observations:

- Earlier formal analysis of ACE-OAuth was incomplete and used Avispa and Scyther, which are tools for automatic verification. These tools are, however, older and less frequently used.
- We hope that extensions to the ACE-OAuth protocol use our model. The added effort for designers of such extensions is relatively small.
- We found a few security aspects that are important to consider: for example, use of multiple audiences needs to be carefully guarded with stronger verification. Some allowed for variants that could be implemented in a compromising manner.
- Most importantly, we have provided the first formal proof that the ACE-OAuth protocol is verifiably secure (albeit if implemented according to requirements, and without improper variations).

7 APPENDIX A: TAMARIN CODE

The code for the generalized ACE-OAuth model, as well as for the ACE-OAuth profile, can be found at:
https://github.com/Yiergot/ACE-OAuth-FormalModel.

REFERENCES

[1] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2014. Easycrypt: A tutorial. In *Foundations of security analysis and design vii*. Springer, 146–166.

[2] Bruno Blanchet. 2003. Automatic verification of cryptographic protocols: a logic programming approach. In *PPDP*, Vol. 3. Citeseer, 1–3.

[3] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (March 1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

[4] Gavin Lowe. 1997. A hierarchy of authentication specifications. In *Computer security foundations workshop, 1997. Proceedings., 10th*. IEEE, 31–43.

[5] Simon Meier. 2013. *Advancing automated security protocol verification*. Ph.D. Dissertation. ETH Zurich.

[6] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*. Springer, 696–701.

[7] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig. 2019. *Authentication and Authorization for Constrained Environments (ACE)*. Internet-Draft draft-ietf-ace-oauth-authz-18. Internet Engineering Task Force. https://tools.ietf.org/html/draft-ietf-ace-oauth-authz-18 Work in progress.

[8] Goldwasser Shafi and Silvio Micali. 1984. Probabilistic encryption. *Journal of computer and system sciences* 28, 2 (1984), 270–299.