Actions Speak Louder Than Passwords:
Dynamic Identity for Machine-to-Machine Communication

Wil Liam Teng
University of Oxford
Oxford, United Kingdom
wil.teng@cs.ox.ac.uk

Kasper Rasmussen
University of Oxford
Oxford, United Kingdom
kasper.rasmussen@cs.ox.ac.uk

ABSTRACT

Machine-to-Machine (M2M) communication is communication between computers without a human user involved. This is a very common paradigm whenever automated tasks are executed routinely, e.g., backup data to a cloud storage, update a local database cache, fetch the latest updates for software, etc. One challenge in this setting is that the credentials to establish secure connections between machines during execution must be available to the machines without any human interaction. Typically that means the credentials must reside on the machine itself, in the form of a secret such as a password, API key, single sign-on token, etc. In practice the secret is often embedded directly into an automatically executed script, but regardless it needs to be stored either in the clear or encrypted with another secret that is available to the machine during execution. This exposes the credentials to anyone who can gain access to the machine. In this paper we present *ActionID*, a scheme that mitigates the problem of credential exposure by making a desired sequence of actions for execution as part of the machine's identity. This way, even if the credentials are exposed, they are only temporarily valid for one particular action sequence that cannot be changed for future executions. We introduce a trusted third party who issues new identities, validates new action requests, and acts as a centralised location for managing access control policies for an arbitrary number of clients and servers. In addition to yielding strong security guarantees, it also simplifies the management of complex access control for an organisation. We present detailed protocols for *ActionID*, along with a thorough security analysis. We implement *ActionID* as a Python library to show the ease of integration into existing applications, and to demonstrate the performance of the scheme, which is on par with SSH.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**; **Authentication**; Authorization; Access control.

KEYWORDS

Machine Identity, Machine-to-Machine (M2M) Communication

1 INTRODUCTION

Machine-to-machine (M2M) communication refers to the communication between machines without the interaction of a human user. M2M communication is used every day by millions of devices for automated and scheduled tasks that require communication with other devices. This includes Content Distribution Networks (CDN); Extract, Transform, Load (ETL) processes; and Development Operations (DevOps) such as automatic backups, automatic software updates, and many, many other things which would otherwise need manual human labour. In addition M2M communication is the foundation for the so-called Internet-of-Things (IoT) as well as most industrial processes. The data aggregation and statistics company Statista Research Department [37] estimates the projected M2M total industry size in 2022 to be worth almost 200 billion US dollars.

One of the central security challenges of M2M communication is how to establish secure communication channels between machines in an efficient and scalable way. With connections being made without human interaction, the secret keys needed to bootstrap a secure channel needs to be available to the machines when needed. That often means the secrets are stored on the machine or device in the clear, or perhaps protected by another key which is itself stored in the clear, which does not provide much more security from a determined attacker. This is a problem because anyone with access to such a machine can extract the credentials and use them to access the service independently from the machine where they were extracted from. In a business organisation, this could be anyone from the IT staff to external adversaries who manage to compromise a machine. In fact, this issue has resulted in many security breaches at GitHub [21], Uber [4], and npm [31].

In this paper we propose *ActionID* which is a novel solution to this problem. It would of course be desirable to entirely prevent credential theft in the first place, but since it is nearly impossible to fully prevent theft from, say, the administrator of the machine of a company, we take a different approach. Rather than trying to prevent credential theft altogether, we make sure that the stolen credentials can only be used for a very specific well-defined purpose, and only for a short amount of time, after which they will have to be renewed. We achieve this by associating the identity of a machine with not only its credentials, but also to every action sequence it wants to execute. Not only does this limit what an adversary with a compromised credential can do, this association further allows

a central entity to monitor the specific usage of each identity, and build up a picture of who is doing what in a company's network. If an adversary is forced to continuously renew stolen credentials, the chances of detecting a credential theft is also increased since the credentials will be renewed more often than before.

ActionID also provides a convenient way to set up and manage user accounts and access control policies for an entire organisation in one place. This is desirable because if access control decisions are made by individual services, then any changes to the access control policies, including adding new users or removing old ones, have to be applied locally to each individual service. *ActionID* enables a unified access control policy to be put in place in a central location, and gracefully handles both identity management and revocation of access. We present the protocols that make all this work efficiently along with a thorough security analysis that prove the necessary security guarantees hold. We also implement *ActionID* as a library to show the performance and scalability of the idea. From that we conclude that even our implementation in Python has a performance on par with other tools that can establish a secure connection, e.g., SSH.

We summarise our main contributions as follows:

- We propose a M2M scheme that utilises a novel machine identity: the assocation of identity with specific actions requested for a specific server. This forms a dynamic machine identity that reduces the implications of a machine compromise and credential theft.
- We design two protocols that make use of this dynamic identity to give strong security guarantees to the principals, even in the presence of a powerful adversary that has the capability to compromise machines.
- We prove that our protocols are secure with respect to the relevant security properties.
- We provide an implementation of *ActionID* as a Python library which makes it easy to deploy on a large class of devices and our implementation is available on GitHub. We use it to document the performance and scalability of the scheme.

This represents a brand new way to think about machine identity, not just as the owner of a private or symmetric key, but as a collection of actions backed up by a traditional identity. If any of the actions changes, that means the identity changes, so any access tokens issued are no longer valid. This effectively limits an adversary that did manage to steal credentials, to do only the same action as the owner of the credential, and only for a brief time, within the limits set out by the access control policies.

2 RELATED WORK

In this section, we discuss the related research areas that inspired *ActionID*, namely, the current machine authentication methods, the state-of-the-art Machine-to-Machine (M2M) authentication protocols, and the developments in Single Sign-On (SSO) systems.

2.1 Machine Authentication Methods

While the authentication methods for a human has been well-established (what you know, what you have, what you are), this is not the case for the authentication methods for a machine. The

authentication methods for a machine can be categorised into three main families: secret-based, context-aware, and hardware-based. Traditionally the most common method to authenticate a machine is through its possession of a secret that can come in different forms. These include passwords, certificates (and the corresponding private keys) [7, 9], API keys [2, 16, 17, 25, 30], and SSO access tokens [22, 28, 36, 38]. Context-aware authentication methods [29] identify a machine based on the measurements of physical features of a machine's surroundings, such as geolocation [10, 42] and proximity [6, 32], which require making assumptions about the physical environment of a machine. Hardware-based authentication methods authenticate a machine by its responses to the challenges issued by a verifier based on a secret derived from the underlying hardware of the machine, e.g., through Trusted Module Platforms (TPMs) [26, 46] or Physical Unclonable Functions (PUFs) [5, 8, 35]. Without making any assumption on a machine's physical environment and its underlying hardware, in this paper we introduce a new type of machine authentication method based on its sequence of actions for execution to mitigate the problem of a credential theft in an organisational network.

2.2 M2M Communication Schemes

As our end goal is to secure M2M connections made between machines while also reducing the dependence on secret information, here we look into some of the existing M2M authentication schemes for various application purposes. The de facto standard for the purpose of a client machine authenticating a server over the web is the Transport Layer Security (TLS) protocol [11]. TLS ensures a client that it is connecting to a server which is the legitimate owner of its public key, and also for establishing a secure channel between the client and the server. For network administrators, the Secure Shell (SSH) protocol [41] is the most common protocol to secure messages exchanged during a remote login from a SSH client to a SSH server. Internet Key Exchange (IKE) protocol [23] is used in Internet Protocol Security (IPSec) for creating secure Virtual Private Networks (VPNs) between two machines. Recently, M2M authentication protocols have been proposed [13, 15, 24, 27, 34, 39] specifically for resource-constrained devices in Internet-of-Things (IoT) networks that put heavy emphasis on the performance of the protocols. Although these M2M authentication schemes are sufficient for their specific use cases, the security of these protocols rests on the assumption that machine credentials are being secret. While this has been the convention for many protocols, as we explain later in Section 3.1, this assumption might not always be fulfilled in every M2M communication scenario. To also account for these additional M2M communication scenarios, we propose *ActionID*, a M2M scheme that limits the implications of a compromised machine credential while also simplifying the credential management for machines. We provide a comparison between *ActionID* with existing M2M communication scheme in Section 3.3.

2.3 SSO Systems

Although not strictly considered as a M2M authentication scheme, we explore Single Sign-On (SSO) systems here because the goals and architecture of SSO systems bear some resemblances to our

work. SSO systems go hand-in-hand with other user authentication mechanisms, and the attractiveness of SSO system stems from the fact that they reduce a user's reliance on having to manage multiple secrets which are essentially passwords. Specifically, SSO systems allow a user to access multiple servers using only a single password, and simultaneously without having to reveal a user's password to the servers. This is done through a trusted third party called the identity provider, whom a user has registered the password with. Kerberos [38] is arguably the first deployed SSO system to secure network services in MIT. Then, a family of SSO systems, OAuth 2.0 Authorisation Framework (OAuth) [22], OpenID Connect (OIDC) [36], Security Assertion Markup Language 2 (SAML2) [28] are deployed for cross-organisational user authentication and authorisation through web browsers. Various research proposals have since been made to enhance the security and privacy of SSO systems. One interesting line of research [1, 3, 33, 43] focuses on distributing the risk of a single identity provider compromise into a number of identity providers to prevent the identity provider from becoming a single point of failure. Another research area focuses on solving the privacy issues of user activity tracking by a curious identity provider or collusive servers. The proposed SSO schemes [14, 18–20, 44, 45] provide user untraceability and unlinkability from the identity provider and servers. This shows that SSO systems have garnered much research attention and are still an active research area. However, where SSO systems excel in user authentication, our focus is on M2M communication in which there is no human user involved. We further compare the differences of *ActionID* with existing SSO systems in more detail in Section 3.3.

3 MOTIVATION AND DESIGN

We first look into the specific problems arising from a typical M2M communication scenario in an organisation's network as our motivation. We then show how existing systems are not designed specifically to handle these problems. To address these problems, our solution is to associate a machine's identity not only to its credentials, but also to its actions and we further list the design goals that we aim to achieve with this idea.

3.1 Motivation

In Figure 1, we illustrate a general abstracted scenario of M2M communication in an organisational network between machines owned by the organisation. Without a human in the loop, a requesting machine, or simply a *requestor*, accesses a *server*'s services by running a *script*. A script typically contains the task to be completed by a server, which we termed as *actions*, and the *credential* a requestor uses to authenticate itself to the intended server. Examples of actions include database scripts, shell commands, or even custom predefined strings readable and executable by a server; examples of credentials are mentioned in Section 2.1. We refer to Figure 8 for a concrete example of a requestor's script. Each server imposes and enforces *access control policies* of who can access what and checks if the execution of a requestor's actions is allowed, e.g., if a requestor's access to a file is authorised.

Without a human to actively input credentials when accessing a server, a requestor's credentials must be available locally to the requestor in the clear, e.g., a hard-coded password embedded in
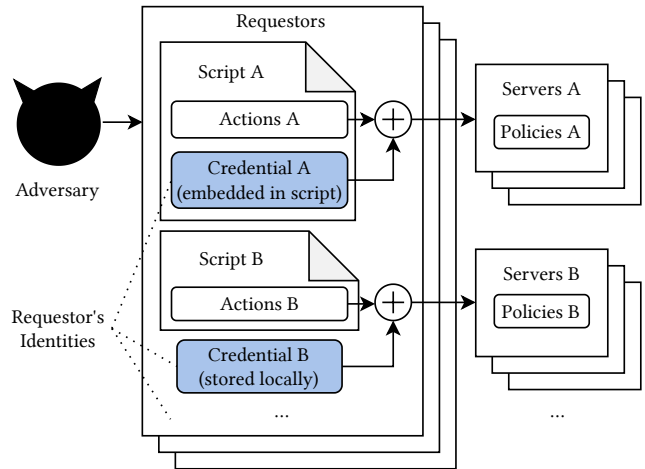
Figure 1: A general abstracted scenario of a M2M communication in an organisational network. A requestor runs scripts specifying actions to be executed and the credential to authenticate to a server. Without a human supplying the credentials, the requestor's credentials are typically stored locally in the clear, e.g., hard-coded in the requestor's script, allowing an adversary who has access to the requestors to extract the credentials. A server enforces access control policies that manage the execution of the requestor's actions. Multiple servers may share the same policies.

the script of the requestor, or a private key stored locally on the disk storage of the requestor. The credentials might in turn be encrypted by other secrets that are stored on a requestor unencrypted. Nonetheless, an adversary having physical access to a requestor can obtain the unencrypted credentials or extract the secrets to decrypt the wrapped credentials. As the credentials represent a requestor's only identity to a server, an adversary with such credentials can impersonate the victim requestor from another machine.

From a management's point of view, this scenario of a M2M communication has several disadvantages. Since the access control policies are managed by individual servers, changes to the access control policies shared by several servers require local changes to the policies at each of the servers. Even with the access control policies in place, mismanagement of a server's access control policies, e.g., resulting from human errors [40], allows for an adversary with compromised credentials to successfully request for the execution of arbitrary actions on behalf of the compromised requestor. As for the credential management of a requestor, each requestor has to manage a unique credential for each service offered by the servers. These management issues can quickly become unscalable for an organisation with a large amount of servers.

On one hand, the problem with relying solely on a secret information to identify machines is that the secret information can be stolen. On the other hand, it is also impossible to establish secure connections and differentiate machines without using any secret information. Ideally it would be perfect if there is a foolproof way of preventing credential theft, but since this is impossible to achieve, we take a different approach: instead of worrying about a piece of

secret information getting stolen, we limit the consequences of a credential theft by restricting what the adversary with the stolen credential can achieve. Building on this approach, we propose a novel machine identity for use in a M2M communication where a machine's identity is not only made up of its credentials but also the actions it wants to execute. With our proposed system model, we simplify the management of such machine identities.

3.2 Design Goals

Our solution to address the problems presented in Section 3.1 is to associate a requestor's identity to not only its credentials, but also its actions. Here we list the five design goals based on this novel idea. The first four design goals focus on the security aspect of our system and the last design goal focuses on the credential and policy management of the system. We later discuss how our system achieves the last design goal in Section 4.1, and prove the four security-related design goals of the system in Section 6.

3.2.1 Mitigate Implications of Credential Leakage. Unlike existing systems where machine credentials are only based on secrets, an adversary with unrestricted access to credentials can impersonate the victim from a different machine. A server is then unable to differentiate between an adversary with a compromised credential from the legitimate machine based solely on a machine's possession of some secret. Our goal here is then for a requestor's actions to become an additional authentication factor for a certain requestor. While no protocol can provide security guarantees against an adversary with unrestricted access to a secret, we can mitigate the implications of such credential leakage. Specifically, we require such an adversary should not execute arbitrary actions but only the same set of actions allowed for the victim requestor. Even if the adversary has both the "valid" actions and the credentials, we further require the actions must first be declared before execution and that the declared actions can only be executed for a short limited time, after which the declaration is again needed. Not only do we restrict what an adversary with compromised credentials can achieve, we provide a way for detection of credential compromise by observing if a requestor's actions deviate from its "normal" actions while also increasing the chances for such detection.

3.2.2 Bind Identity to Actions. Based on our requirement of requestors declaring actions ahead of time, the declared actions essentially forms a part of the requestor's identity for its session with the server. An adversary with compromised credentials might attempt to circumvent the requirement by not declaring its actions at all, or by first declaring "valid" actions, and then sending undeclared actions for execution. To avoid the adversary from arbitrarily changing its identity using undeclared actions, we thus need a mechanism to detect if the integrity of a requestor's identity has been violated, i.e., if the declared actions of a requestor has changed. Additionally we need a way to detect if an adversary tries to evade detection by executing already declared actions beyond their limited time, i.e., extending the expiration time of the already declared actions.

3.2.3 Liveness and Secure Channel Establishment. To prevent man-in-the-middle attacks and replay attacks, we additionally necessitate mutual liveness checks to be carried out between a requestor and a server before the execution of the requestor's actions. Depending on the specific application, the execution might involve further exchange of session data between the requestor and the server. We would thus want to prevent the session data from being sent as cleartext since the session data might contain sensitive information. The secure channel would additionally need to ensure the integrity of the session data to prevent its modification.

3.2.4 Confidentiality of Actions. A requestor's actions might contain sensitive application data about the system such as the contents of a file or the entries of a database. Therefore even the actions has to be kept secret from anyone other than the requestor's intended recipients when a requestor sends its actions for declaration or execution. To prevent the sensitive data from being leaked during a transmission, we thus require the confidentiality of a requestor's actions so that only intended recipients of a requestor can retrieve its actions. Although this does not stop an adversary with physical access to a requestor from obtaining the requestor's actions, the only thing the adversary can do with stolen actions is essentially to execute the actions on the victim requestor's behalf. Since the actions are already allowed to be executed by the victim requestor, the implication for the stolen actions is thus minimised.

3.2.5 Simplified and Centralised Management. Our desire is that the access control policies of the servers are to be easily managed as the decision for the authorisation of a requestor's actions involves directly the management of these policies. Specifically, a change in the policies shared by several servers should only be applied once, instead of the same changes being applied multiple times at each individual server. This creates a more centralised policy management desirable especially in an organisation with a large number of servers. Additionally, as a requestor's actions are already specific to a particular server, we will not require a requestor to manage a unique long-term credential for each server. We would thus aim for a requestor to have only a single long-term credential, paired with its actions to form a disposable short-term credential. This, in turn, reduces the number of long-term credentials that a requestor has to manage to only one, regardless of the number of server present in the system. Since the requestor has to manage its actions anyway regardless of the number of its long-term credentials, our design goal, if fulfilled, eases the credential management of a requestor.

3.3 Comparison with Existing Systems

The M2M scenario described in Section 3.1 is a very common setting for any organisation running automated tasks and thus it is surprising that existing systems have still yet to address these problems. As our focus is machine authentication, the de facto standard for authenticating machine and establishing secure channels between machines such as SSH [41] and TLS [11] are still not adequate for our M2M setting. This is because the security of these protocols require the assumption that the credential of a machine, i.e., a private key, being secret and they do not consider the presence of an all-powerful adversary having unrestricted access to the credentials of a machine. Similarly, SSO systems and the research work on SSO systems assume one or more secret information such as passwords [1, 3, 33, 38, 43] and access tokens [22, 28, 36], which, if compromised, can lead to impersonation attacks as seen from real-world scenarios [4, 21, 31]. Additionally, widely deployed SSO
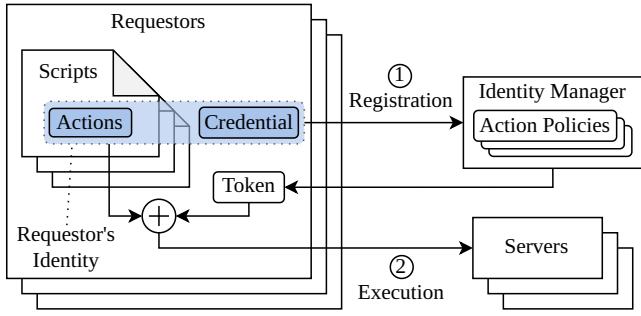
Figure 2: An overview of *ActionID*. A requestor's identity comprises of both its actions and its long-term credential. The identity manager is a trusted entity who manages all servers' action policies. A requestor registers its identity ahead of time to get a token for the execution of its actions on a server.

Figure 3: An overview of the system and adversary model for *ActionID*. The system model consists of requestors, servers, and the trusted identity manager. The adversary model consists of a malicious requestor and an external adversary. Both adversaries have full control of the communication channel.

systems such as Kerberos [38], OAuth [22], OIDC [36], SAML2 [28] are designed for user authentication. This means that the presence of a user and user interaction, in one form or another, e.g., typing in passwords or completing Multi-Factor Authentication (MFA) challenges, are needed during service access to a machine. These SSO systems are thus not appropriate for application in a M2M setting where there is no human user involved. We emphasise that our goal here is not to replace existing systems and protocols, but rather, since existing systems do not address the more specific security and management issues that stem from an organisation's internal M2M communication processes as described in Section 3.1, we thus propose a new system that leverages this idea of a machine's actions as part of its identity to specifically deal with these issues.

4 OVERVIEW AND MODELS

We give an overview of the design of *ActionID* in this section. We later provide the system model and the adversary model in detail, which include the assumptions made for the entities in the system and also for the adversaries.

4.1 *ActionID* Overview

We propose *ActionID*, a novel M2M scheme for an organisation's network where a requestor's identity comprises of both its credential and actions. A high-level overview of *ActionID* is provided in Figure 2. We introduce a trusted third party called the identity manager who centrally stores and manages all *action policies* of servers. Action policies here refer to more than just authorisation policies (who can access what), but also fine-grained or even application-specific conditions for the execution of actions to happen. For example, some actions should be executed only at a certain day of the week and are only executed for a fixed number of times in a certain time period. As the policies are centrally stored at the identity manager, we have now achieved the last design goal in Section 3.2 as changes to the policies shared by several servers are only applied once at the identity manager, instead of being individually applied at each server.

For executing actions, we require that a requestor first declares its actions ahead of time by registering them with the identity
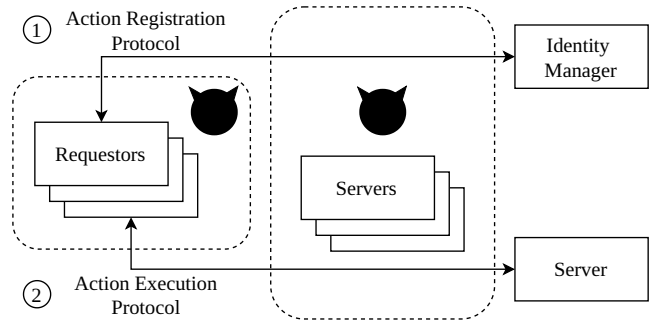
manager. If the actions of a requestor comply with the policies of its intended server, the identity manager issues a short-lived token to the requestor to be presented to the server for the execution. Our design not only forces an adversary with compromised credentials to always reveal its intended actions for execution, but also helps the identity manager in monitoring for credential compromise and building up a picture of who does what on the network by having a centralised log of registered actions. This can further facilitate anomaly detection on the logged actions to determine if a requestor's actions deviate from its "normal" actions, which could be an indication of a compromised requestor. The short-lived token limits the time window for action execution. So, even if an adversary's actions comply the policies, the adversary can only execute specific actions only for an allowed duration of time, after which the registration of actions is again needed, limiting the impact of a credential leakage and increasing the chances of detection.

Although the identity manager poses the risk of being a single point of failure, in practice trusted third parties are still being used in organisational networks as trust anchors for the security of organisational systems, despite of these risks. Examples include the Key Distribution Centre in Kerberos [38], and identity providers in OAuth [22] and OIDC [36]. While a centralised identity manager might raise privacy concerns, privacy of machines from the organisation that owns them is generally undesirable in an organisational network, e.g., a corporate network, where it is, in fact, more preferable for organisations to be aware of who is doing what.

4.2 System Model

As depicted in Figure 3, our system model consists of three entities: the requestors, the identity manager, and the servers. Each entity possesses a public-private key pair as their long-term credential and has a copy of the public key of the identity manager. All entities communicate in a M2M manner without any human interaction.

In addition to its actions, a requestor possesses a certificate signed by the identity manager and the corresponding private key. The certificate shows that the requestor has previously been registered by the identity manager without the identity manager having to explicitly remember the public key of a requestor.

Each server is previously registered with the identity manager and a copy of the public key of each server is stored by the identity manager. A server is associated with its action policies and trusts that the identity manager enforces the server's policies when issuing a token to a requestor. Only when it is offering its services to a requestor should a server be online.

The identity manager is always online. In addition to managing the public key of each server, the identity manager stores and enforces all action policies of all servers. We do not restrict the implementation details of such policies, e.g., the encoding or the format. Our only requirement is that the identity manager understands the policies' semantics to perform checks for the compliance of a requestor's actions with the policies of the requested server.

4.3 Adversary Model

Also depicted in Figure 3, our adversary model consists of two adversaries: an internal adversary called a malicious requestor, and an external adversary. We model both adversaries as Dolev-Yao adversaries [12] and we assume that the adversaries are incapable of breaking the underlying cryptographic primitives. We also do not consider cases involving trivial Denial-of-Service (DoS) attacks where the adversaries simply drop all communicated messages.

A malicious requestor additionally possesses its own legitimate certificate issued by the identity manager with the corresponding private key. A malicious requestor can also compromise other requestors and has access to their credentials, i.e., their private keys. A malicious requestor has two goals. First, to access a server with actions not already authorised to compromised requestors under its control. The second is for a server to execute unregistered actions.

We further allow an external adversary to collude with other servers in the system where it has access to the private keys of these servers. The only exception to this is the private key of an honest server who is contacted by a requestor for action execution. An external adversary has two goals. The first is to impersonate as either the identity manager or an honest server to a requestor. The second is to break the confidentiality of a requestor's actions.

5 PROTOCOLS DESCRIPTION

We introduce two protocols for *ActionID* to illustrate an example of utilising this novel idea of a machine identity associated with its actions, namely, the Action Registration Protocol and the Action Execution Protocol. A requestor runs both protocols in sequence for every action execution. We further assume the cryptographic primitives used are secure as per their own security definitions.

5.1 Action Registration Protocol

Figure 4 depicts the Action Registration Protocol. A requestor runs this protocol to register its actions to the identity manager and to receive a fresh token on its registered actions.

A requestor initiates the protocol by sending message m_1 containing a nonce n, its actions A, and the identifier of its intended server S. The nonce and the actions are encrypted to, respectively, serve as a liveness challenge for the identity manager and ensure the confidentiality of the actions. Message m_1 is sent together with the requestor's certificate and the requestor's signature on the message.
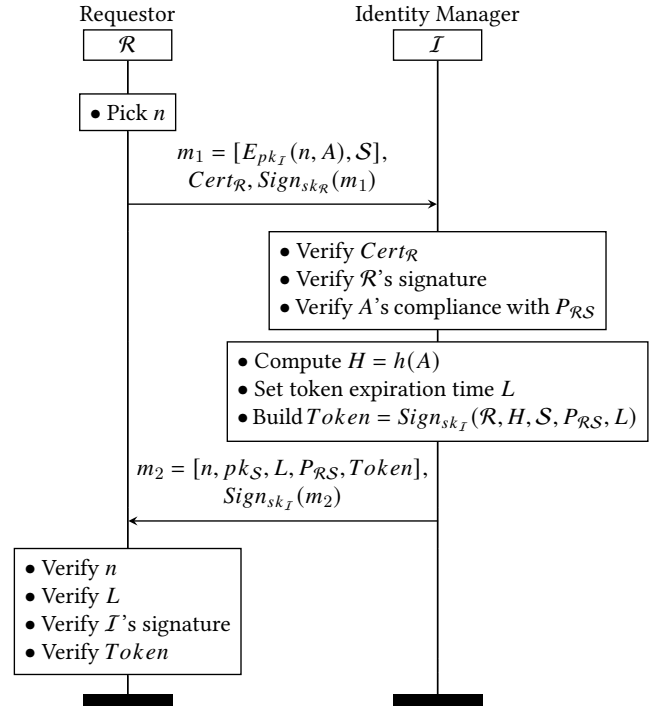
Figure 4: The Action Registration Protocol involving a requestor $\mathcal{R}$ and the identity manager $\mathcal{I}$. The requestor initiates the protocol to register its actions and to obtain a token from the identity manager. The token is required for the later execution of the requestor's actions.

After verifying the certificate and the signature, the identity manager checks if the requestor's actions comply with the action policies of the requested server, denoted as $P_{\mathcal{RS}}$. We give an example of how the identity manager completes this task in more detail in Section 7.2. If so, the identity manager proceeds to generate a token by first hashing the actions and setting an expiration time for the token. The identity manager then hashes five information, namely, the expiration time of the token L, the policies of the server $P_{\mathcal{RS}}$, the hashed actions H, and the identifiers of the requestor $\mathcal{R}$ and the server S, and signs the resulting hash to construct the token.

The identity manager further hashes, signs, and sends a reply message m_2 to the requestor, which encompasses the token and the information needed to verify the token, along with the requestor's nonce and the public key of the server. The requestor subsequently verifies the nonce, the expiration time of the token, the signature of the identity manager on message m_2, and the token.

The token is stored by the requestor for action execution and it will not have to register the same actions if the token has not expired. The requestor can simply discard an expired token and request for a new token by repeating the Action Registration Protocol.

5.2 Action Execution Protocol

Figure 5 illustrates the Action Execution Protocol. The purpose for this protocol is for the requestor's registered actions to be executed
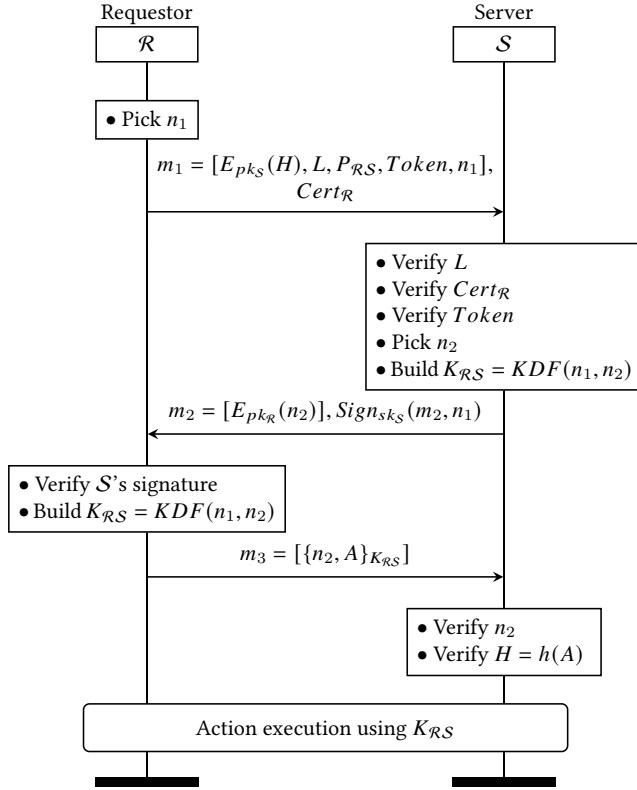
Figure 5: The Action Execution Protocol between a requestor $\mathcal{R}$ and a server $\mathcal{S}$. The requestor initiates the handshake where a fresh session key is agreed. The session key is used for establishing a secure channel for the exchange of subsequent messages during the session resulting from the execution of actions.

by the server. The action execution is further secured by the secure channel established at the end of the handshake of the protocol.

The protocol starts with a handshake where the requestor first commits a nonce n_1 and the hash of its actions to the server. These are sent along with token and the requestor's certificate for the server to verify. To prevent offline guessing attacks against the requestor's actions, the hash of the actions is encrypted.

The server first verifies the token and the certificate. If they are valid, the server encrypts and commits another nonce n_2 to the requestor. The server then signs the ciphertext and nonce n_1 to prove the server's liveness to the requestor, and then computes a fresh session key using a key derivation function $KDF(\cdot, \cdot)$ with the two nonces as input.

After verifying the signature and decrypting nonce n_2, the requestor computes the session key in the same way as the server. The requestor then reveals its actions with nonce n_2 to the server, both of which are encrypted with the freshly established session key. The server verifies both values and if the verifications passed, this guarantees the server two things: the liveness of the requestor, and that the revealed actions are the same as the registered actions.

Once the handshake is completed, the server executes the verified actions of the requestor. Depending on the application, the session could involve exchanging further messages between the requestor and the server. To preserve the confidentiality and integrity of the session messages, they are to be secured with the established session key, which forms a secure channel between the requestor and the server. The protocol ends when the session is terminated.

6 SECURITY ANALYSIS

We now provide the security guarantees for *ActionID* with respect to the design goals in Section 3.2 and we informally prove these security guarantees by contradiction. For each proof we first exhaustively enumerate the scenarios in which an adversary, either a malicious requestor or an external adversary, can choose to break the guarantee, and prove that all scenarios contradict our assumptions. We emphasise that the underlying cryptographic primitives used in the protocols as described in Section 5 are secure.

GUARANTEE 1 (MITIGATE IMPLICATIONS OF CREDENTIAL LEAKAGE). *A malicious requestor in possession of private keys of requestors can get tokens only for actions authorised to the victim requestors according to the action policies and can execute the actions only for a limited period of time.*

PROOF (SKETCH). For a malicious requestor to break this guarantee, the adversary must possess a token without a restricted set of actions, i.e., without complying with the action policies, or with no expiration time. There are three ways a malicious requestor can achieve this. (i) First, the adversary can attempt to manipulate the activity of the identity manager to not follow the Action Registration Protocol for issuing tokens. However, in successfully doing so, this breaks our system model where the identity manager is trusted. (ii) Second, the adversary can compromise the private key of a requestor whose set of authorised actions contain the adversary's chosen actions, and use that credential to get a token. This allows the adversary to register new actions and continuously renew tokens, and thus the guarantee may seem to be broken. However, as the token is issued only when the chosen actions comply with the policies for the newly compromised requestor, this means the adversary still has to choose from a restricted set of actions, and so the guarantee still stands. (iii) Third, to have complete control over the token, the adversary can forge the token for any arbitrarily chosen actions or expiration time of the token. This requires the adversary to possess the private key of the identity manager, and this can only happen in two ways: either the adversary breaks the underlying signature scheme, or guesses the private key. The first is impossible by our assumption of secure cryptographic primitives and the second is only possible with a negligible probability. □

GUARANTEE 2 (BIND IDENTITY TO ACTIONS). *A requestor must first register its actions, and once registered, the requestor cannot change the registered actions.*

PROOF (SKETCH). A malicious requestor can break this guarantee if (i) it executes the actions without registering, or (ii) by changing the registered actions. (i) To successfully execute actions without registration, a malicious requestor needs to present a token for message m_1 in the Action Execution Protocol. The adversary can

either forge or replay a token. Forging a token is impossible for the reasons discussed in Guarantee 1. If the adversary replays a token, this means that the list of actions associated with the token has already been previously registered. Thus, the first part of the guarantee holds. (ii) Even after registering, a malicious requestor might still attempt to cheat by revealing actions that are different from the registered actions in message m_3 of the Action Execution Protocol. To bypass a server's verification for the modified actions, the modified actions has to produce the same hash as the registered actions, which is equivalent to searching for a hash collision. This is only possible with negligible probability as we require the assumption that a cryptographically secure hash function is used. Thus the second part of the guarantee holds.   □

GUARANTEE 3 (LIVENESS AND SECURE CHANNEL ESTABLISHMENT). *A requestor and a server mutually check each other for liveness and subsequently establish a fresh session key known only to them for action execution.*

PROOF (SKETCH). For an external adversary to break this guarantee, the adversary must establish the session key in the Action Execution Protocol when a requestor contacts its intended server. The adversary has three options: (i) impersonate as a server contacted by a requestor, (ii) impersonate as the identity manager to a requestor, or (iii) impersonate as a requestor to an honest server. (i) In the first case where the adversary impersonates as an honest server, the adversary has to convince the requestor by generating a valid signature of the honest server in message m_2 of the Action Execution Protocol. This requires the adversary to own the private key of the honest server. (ii) The adversary can circumvent this requirement in the second case. The adversary modifies message m_2 of the Action Registration Protocol to change the honest server's public key with the adversary's own public key, allowing the adversary to generate the signature in message m_2 in the Action Execution Protocol with the adversary's own private key. In this case, however, the adversary has to forge the identity manager's signature on message m_2 of the Action Registration Protocol and impersonate the identity manager by having the identity manager's private key. (iii) In the third case where the adversary impersonates as a requestor to an honest server, the adversary has to decrypt the challenge n_2 issued by the honest server in message m_2 of the Action Execution Protocol, which again requires the adversary to possess the requestor's private key. For these three impersonation attacks by the adversary to succeed, the adversary must be in possession of a private key of some sort. As we have seen before, the adversary can only either break the underlying cryptographic primitive, or guess the private key (or the challenge nonce). The former infringes on our assumption of secure cryptographic primitives. The latter, similar to brute-forcing a session key and guessing a nonce challenge, can only occur with a negligible probability.   □

GUARANTEE 4 (CONFIDENTIALITY OF ACTIONS). *A requestor's actions must be known only to the identity manager for registration, and only to the intended server for execution.*

PROOF (SKETCH). Breaking this guarantee means an external adversary can retrieve a requestor's actions. Since actions (and the corresponding hash) are encrypted in both protocols, the adversary
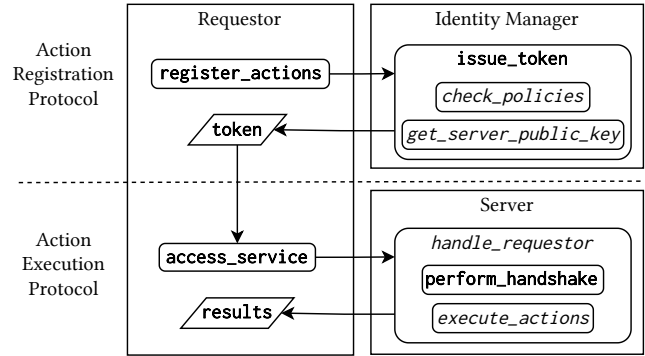
Figure 6: An overview of the implementation for *ActionID*. Rounded-corner rectangles are functions labelled with the name of the function. Function names in bold are functions provided in `actionid.py` whereas function names in italics are application-specific functions.

has two methods of achieving this: (i) decrypting the encrypted actions and subsequently launch offline guessing attacks, or (ii) deriving the session key. (i) For the first method, the adversary must decrypt the ciphertext in message m_1 from one of the two protocols. If the adversary chooses to decrypt the encrypted hash from message m_1 of the Action Execution Protocol, the hash is brute-forced to further guess the requestor's actions in an offline manner. However, the first method requires the adversary to be in possession of the private key of the identity manager or the server, and this is only possible if the adversary either breaks the underlying encryption scheme or guesses the private key. As we have seen many times, the former violates our assumption that secure encryption schemes are used while the probability of the latter happening is negligible. (ii) For the second method, the adversary must decrypt the actions encrypted in message m_3 of the Action Execution Protocol by deriving the session key. As we have proved in GUARANTEE 3, the adversary successfully deriving the session key leads to the violation of our assumptions.   □

7 IMPLEMENTATION

To demonstrate the practicality of our scheme, we provide an implementation as a proof-of-concept for *ActionID*. Based on the implementation, we conduct experiments to test the performance of *ActionID* against a well-known protocol: SSH. This section covers the findings for the implementation as well as the experiments.

7.1 Overview

We implement the proof-of-concept for *ActionID* in Python and it comprises of two parts: a Python library, `actionid.py`, and a working example for `actionid.py`. Our implementation imagines an application scenario where a requestor routinely uploads and retrieves data to and fro a SQL server. Figure 6 illustrates the interactions of the four main functions provided in `actionid.py`. A requestor registers and executes actions via `register_actions` and `access_service` respectively. The identity manager completes the Action Registration Protocol and issues tokens via `issue_token`,

```
{
    "any":
    {
        "blocked_actions": ["DROP", "DELETE"]
    }
}
```

Figure 7: The action policy encoded in the JSON format for the proof-of-concept. The policy specifies the SQL actions not permitted for requestors under the user group any.

whereas the server performs the handshake of the Action Execution Protocol via `perform_handshake`. To accommodate for various application needs, we further demonstrate that `actionid.py` works with application-specific functions. In our implementation, these functions are `check_policies`, `get_server_public_key`, `handle_requestor`, and `execute_actions`. Our proof-of-concept is available on GitHub[1] for interested readers.

7.2 Identity Manager

We provide a more concrete idea on how the identity manager fulfills its responsibilities in the Action Registration Protocol, mainly, how the identity manager (a) checks the compliance of a requestor's actions with the server's action policies (`check_policies`) and (b) stores the public key of servers. (a) The main responsibility of the identity manager, i.e., analysing the compliance of the requestor's actions, requires the consideration of three design choices: (i) the encoding of the server's action policies, (ii) the encoding of the requestor's actions, and (iii) the mechanism for analysing the compliance of the requestor's actions against the action policies. (i) We choose the encoding of a server's action policies to be in the JSON format and as shown in Figure 7, it represents a blocklist of the SQL actions that the requestors under the user group any are prohibited to execute. (ii) As the server is a SQL database server, the requestor's actions are thus encoded in SQL and should be comprised of syntactically valid SQL statements. (iii) Finally, for the identity manager to analyse a requestor's actions so as to enforce the action policies, the identity manager simply searches for the presence of the prohibited SQL keywords in the string of the requestor's actions, i.e., SQL statements. We acknowledge that static analysis of arbitrary statements for a Turing-complete programming language is undecidable in general. However, as our implementation shows, static analysis with the purpose of access control can be done by methods as simple as searching for forbidden strings. Indeed, static analysis of a requestor's actions can further be implemented using other more complicated parsing methods. (b) If the actions of a requestor comply with the policies, the identity manager computes the token and subsequently retrieves the public key of the requested server (`get_server_public_key`) from its database. Our implementation uses a SQL database for the identity manager for storing and managing the public key of the servers.

7.3 Server

We give a more detailed explanation here on how a server handles a connection from a requestor in the Action Execution Protocol. A server first performs the handshake (`perform_handshake`) with

[1]https://github.com/wilteng/actionid

```
import actionid

# Network configurations
IDM_IP = "127.0.0.1"
IDM_PORT = 8081
SERVER_IP = "127.0.0.1"
SERVER_PORT = 8082
server_id = '81258728'

# Execute actions for data pull
actions = 
"""
SELECT * FROM fruits;
"""
token = actionid.register_actions(IDM_IP, IDM_PORT, server_id, actions)
results = actionid.access_service(SERVER_IP, SERVER_PORT, actons, token)
print('\n"fruits" table:' + results)

# Execute actions for data push and pull
actions = 
"""
INSERT INTO fruits ("fruit", "colour") VALUES ("grapes", "purple");
SELECT * FROM fruits;
"""
token = actionid.register_actions(IDM_IP, IDM_PORT, server_id, actions)
results = actionid.access_service(SERVER_IP, SERVER_PORT, actions, token)
print('\n"fruits" table:' + results)
```

Figure 8: An example of the requestor's script that uses `actionid.py`. The requestor's script uses `register_actions` and `access_service` functions from `actionid.py` to retrieve from and insert to the database of the SQL server.

a requestor. After a successful handshake, the server receives the registered actions from the requestor and proceeds to execute the actions (`execute_actions`). Similarly, the encoding of the requestor's actions remains an important design choice for the server's implementation, as it should recognise and understand the semantics of the requestor's actions to execute them. Since the server is an SQL server in our implementation, the encoding of the requestor's actions should obviously be syntactically valid SQL statements. The action execution ends after the results have been returned to the requestor encrypted with the session key and appended with a Message Authentication Code (MAC). Depending on the application, the action execution can be implemented in alternative ways. For example, both the requestor and the server can choose to keep the connection alive and continue exchanging messages after the handshake. The confidentiality and the integrity of subsequent messages exchanged during this time can still be secured with the session key.

7.4 Requestor's Script

Figure 8 shows a concrete working example of a requestor's Python script in our implementation. The script shows the requestor completing two sets of SQL operations. The result from running the requestor's script is shown in Figure 9, which implies a successful execution. Note that since the credential management of the requestor is implicitly handled by *ActionID*, the requestor's script does not specify any kind of credentials, e.g., through statements like `password = "password123"`. This reduces unintentional credentials leakage through the requestor's script, which aligns with our motivation for this paper. Each task of the requestor only requires three lines of Python statements. As colour-coded in Figure 8, the requestor first defines the actions (red statements), then the requestor registers its actions to get a token (blue statements), and lastly the requestor executes its actions (purple statements).

Figure 9: Output of the `requestor.py` script that shows the proof-of-concept for *ActionID* successfully executes the SQL actions that are requested by the requestor.

7.5 Performance Overhead

To test for the feasibility and performance of the *ActionID*, the implementation is compared with the Secure Shell (SSH) protocol [41] as a performance benchmark. Our experiments measure and compare the performance of the requestors to complete the two protocols of *ActionID* versus the performance of a SSH client to transfer its actions over the SSH protocol.

We conduct the experiments in a network-controlled environment consisting of two individual machines where one acts as a requestor, and another acts as both the identity manager and the server. The requestor machine is equipped with an Intel® Core™ i7-9750H processor and the Windows 10 operating system. The identity manager/server machine is equipeed with a processor of Intel® Core™ i5-1145G7 and the Ubuntu 20.04 operating system. The processing time for each requestor to complete both protocols is measured. Similarly, the experiments for testing the performance of SSH protocol are set up in the same network environment using the same machines. The Windows machine acts as a SSH client and the Ubuntu machine acts as a SSH server. The performance of the SSH protocol is similarly measured by the execution time for a SSH client to connect to the SSH server via the `ssh` command.

For interested readers and results reproducibility, we list the parameters and the algorithms of the cryptographic primitives used in the implementation and the experiments. The cryptographic primitives are from the `pyca/cryptography` Python library version 37.0.4. The hash function used is SHA-256, and for asymmetric cryptographic algorithms, RSA with a key size of 2048 bits and an exponent of 65537 is used. For symmetric key algorithms, the block cipher AES-128 in CBC mode is used with a key size of 128 bits and for MAC generation, a Hash-based MAC (HMAC) with a key size of 256 bits is used. The derivation of session key in the Action Execution Protocol (see Section 5.2) is realised using the Concatenation Key Derivation Function (ConcatKDFHash) with the seed being the hash of the two nonces, n_1 and n_2. For experiments involving SSH connections, the authentication of SSH connections is made through a RSA keypair with a key size of 2048 bits generated with the `ssh-keygen` command during experiment setup.

The experiment results are visualised in Figure 10. The figure depicts the distribution of the protocol execution time for 10000 requestors and 10000 SSH clients. Note that the generated data take into account the latency of the network but not the actual execution of a requestor's actions, so only the performance of the protocols are tested. The results show that even as a proof-of-concept, the performance of *ActionID* is comparable and fares quite well with
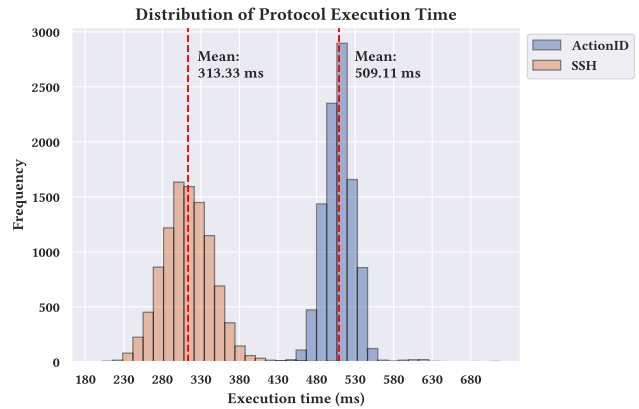

Figure 10: The distribution of the time taken by 10000 requestors to execute the two protocols of *ActionID* and the distribution of the time taken by 10000 SSH clients to connect and transmit its actions to a SSH server. The experimental results shows that *ActionID* has a comparable performance with the SSH protocol.

the performance of a mature and optimised security protocol like the SSH protocol, in spite of *ActionID* requiring more expensive cryptographic primitives, i.e., signature generation and verification.

8 CONCLUSION

The identity of a machine in the context of protocol execution has traditionally been associated with just the credentials of that machine, i.e., knowledge of a private key, or more generally a secret string. However, this means that anyone obtaining these secrets is able to assume the identity of that machine.

ActionID is a scheme that extends the idea of the identity of a machine to the actions that a machine can take when accessing a remote service in an organisation's network. We have shown that with a trusted third party, we can bind the long-term identity of a machine with a list of actions allows us to create a "dynamic identity" that limits the otherwise significant consequences of a theft of credentials. It prevents an adversary with stolen credentials from performing arbitrary actions on the server where those credentials are used. It forces the adversary to always declare and register its intended actions to get a service access token, which helps to increase the chances of detection of and identify stolen credentials.

The use of a trusted third party in *ActionID* also provides a mechanism for the management of user accounts and access control policies of an arbitrary number of clients and servers. These are centrally stored and enforced thereby reducing the chances of the mismanagement of policies, e.g., forgetting to update a particular machine, which, in practice contributes enormously towards strengthening the operational security in an organisation.

Our implementation is in the form of a Python library. We use it to show the ease of integration into existing applications, as well as demonstrate the practical performance of the scheme when we compare *ActionID* to a mature and optimised protocol such as SSH.

9 ACKNOWLDGEMENTS

REFERENCES

[1] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. 2018. PASTA: password-based threshold authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2042–2059.

[2] aws. 2022. Creating and using usage plans with API keys. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html.

[3] Carsten Baum, Tore Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. 2020. PESTO: proactively secure distributed single sign-on, or how to trust a hacked server. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 587–606.

[4] Matt Binder. 2022. A teen hacked Uber and announced it in the company Slack. Employees thought it was a joke. Available: https://mashable.com/article/uber-teen-hacker-slack-joke.

[5] An Braeken. 2018. PUF based authentication protocol for IoT. *Symmetry* 10, 8 (2018), 352.

[6] Stefan Brands and David Chaum. 1994. Distance-bounding protocols. In *Advances in Cryptology—EUROCRYPT'93: Workshop on the Theory and Application of Cryptographic Techniques Lofthus, Norway, May 23–27, 1993 Proceedings 12*. Springer, 344–359.

[7] Matthew Campagna. 2013. SEC 4: Elliptic curve Qu-Vanstone implicit certificate scheme (ECQV). *Standards for Efficient Cryptography, Version* 1 (2013).

[8] Urbi Chatterjee, Vidya Govindan, Rajat Sadhukhan, Debdeep Mukhopadhyay, Rajat Subhra Chakraborty, Debashis Mahata, and Mukesh M Prabhu. 2018. Building PUF based authentication and key exchange protocol for IoT without explicit CRPs in verifier database. *IEEE transactions on dependable and secure computing* 16, 3 (2018), 424–437.

[9] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and William Polk. 2008. *Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile*. Technical Report.

[10] Dorothy E Denning and Peter F MacDoran. 1996. Location-based authentication: Grounding cyberspace for better security. *Computer Fraud & Security* 1996, 2 (1996), 12–16.

[11] Tim Dierks and Eric Rescorla. 2008. *The transport layer security (TLS) protocol version 1.2*. Technical Report.

[12] Danny Dolev and Andrew Yao. 1983. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (1983), 198–208.

[13] Alireza Esfahani, Georgios Mantas, Rainer Matischek, Firooz B Saghezchi, Jonathan Rodriguez, Ani Bicaku, Silia Maksuti, Markus G Tauber, Christoph Schmittner, and Joaquim Bastos. 2017. A lightweight authentication mechanism for M2M communications in industrial IoT environment. *IEEE Internet of Things Journal* 6, 1 (2017), 288–296.

[14] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2015. Spresso: A secure, privacy-respecting single sign-on system for the web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1358–1369.

[15] Lijun Gao, Lu Zhang, Lin Feng, and Maode Ma. 2020. An efficient secure authentication and key establishment scheme for M2M communication in 6LoWPAN in unattended scenarios. *Wireless Personal Communications* 115, 2 (2020), 1603–1621.

[16] GitHub Docs. [n.d.]. Creating a personal access token. Available: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token.

[17] Google Cloud. 2022. Authenticate using API keys. Available: https://cloud.google.com/docs/authentication/api-keys.

[18] Chengqian Guo, Jingqiang Lin, Quanwei Cai, Fengjun Li, Qiongxiao Wang, Jiwu Jing, Bin Zhao, and Wei Wang. 2021. UPPRESSO: Untraceable and Unlinkable Privacy-PREserving Single Sign-On Services. *arXiv preprint arXiv:2110.10396* (2021).

[19] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. 2018. Anonymous single-sign-on for n designated services with traceability. In *European Symposium on Research in Computer Security*. Springer, 470–490.

[20] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, Stephan Wesemeyer, and Nick Wilson. 2019. Anonymous single sign-on with proxy re-verification. *IEEE Transactions on Information Forensics and Security* 15 (2019), 223–236.

[21] Mike Hanley. 2022. Security alert: Attack campaign involving stolen OAuth user tokens issued to two third-party integrators. Available: https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/.

[22] Dick Hardt. 2012. *The OAuth 2.0 authorization framework*. Technical Report.

[23] Dan Harkins and Dave Carrel. 1998. *The internet key exchange (IKE)*. Technical Report.

[24] Hassen Redwan Hussen, Gebere Akele Tizazu, Miao Ting, Taekkyeun Lee, Youngjun Choi, and Ki-Hyung Kim. 2013. SAKES: Secure authentication and key establishment scheme for M2M communication in the IP-based wireless sensor network (6L0WPAN). In *2013 Fifth international conference on ubiquitous and future networks (ICUFN)*. IEEE, 246–251.

[25] IBM. 2022. Creating an IBM Cloud API key. Available: https://www.ibm.com/docs/en/app-connect/container?topic=servers-creating-cloud-api-key.

[26] Issa Khalil, Zuochao Dou, and Abdallah Khreishah. 2015. TPM-based authentication mechanism for apache hadoop. In *International Conference on Security and Privacy in Communication Networks: 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I 10*. Springer, 105–122.

[27] Evangelina Lara, Leocundo Aguilar, Mauricio A Sanchez, and Jesús A García. 2020. Lightweight authentication protocol for M2M communications of resource-constrained devices in industrial Internet of Things. *Sensors* 20, 2 (2020), 501.

[28] Hal Lockhart and B Campbell. 2008. Security assertion markup language (saml) v2. 0 technical overview. *OASIS Committee Draft* 2 (2008), 94–106.

[29] Moritz Loske, Lukas Rothe, and Dominik G Gertler. 2019. Context-aware authentication: State-of-the-art evaluation and adaption to the IIoT. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE, 64–69.

[30] Microsoft. 2022. Use API keys for Azure Cognitive Search authentication. Available: https://learn.microsoft.com/en-us/azure/search/search-security-api-keys#what-is-an-api-key.

[31] Greg Ose. 2022. npm security update: Attack campaign using stolen OAuth tokens. Available: https://github.blog/2022-05-26-npm-security-update-oauth-tokens/.

[32] Kasper Bonne Rasmussen and Srdjan Capkun. 2010. Realization of RF Distance Bounding.. In *USENIX security symposium*. 389–402.

[33] Rachit Rawat and Mahabir Prasad Jhanwar. 2020. PAS-TA-U: PASsword-Based Threshold Authentication with Password Update. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 25–45.

[34] KM Renuka, Saru Kumari, Dongning Zhao, and Li Li. 2019. Design of a secure password-based authentication scheme for M2M networks in IoT enabled cyber-physical systems. *IEEE Access* 7 (2019), 51014–51027.

[35] Ulrich Rührmair. 2022. Secret-free security: A survey and tutorial. *Journal of Cryptographic Engineering* 12, 4 (2022), 387–412.

[36] Natsuhiko Sakimura, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. 2014. Openid connect core 1.0. *The OpenID Foundation* (2014), S3.

[37] Statista Research Department. 2022. M2M (machine-to-machine) – Statistics & Facts. https://www.statista.com/topics/1843/m2m-machine-to-machine/.

[38] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. 1988. Kerberos: An Authentication Service for Open Network Systems. In *IN USENIX CONFERENCE PROCEEDINGS*. 191–202.

[39] Chalee Thammarat and Chian Techapanupreeda. 2021. A secure authentication and key exchange protocol for M2M communication. In *2021 9th International Electrical Engineering Congress (iEECON)*. IEEE, 456–459.

[40] DBIR Verizon. 2020. Data Breach Investigations Report 2020. *Computer Fraud & Security* 4 (2020), 30059–2.

[41] Tatu Ylonen and Chris Lonvick. 2006. *The secure shell (SSH) transport layer protocol*. Technical Report.

[42] Feng Zhang, Aron Kondoro, and Sead Muftic. 2012. Location-based authentication and authorization using smart phones. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 1285–1292.

[43] Yuan Zhang, Chunxiang Xu, Hongwei Li, Kan Yang, Nan Cheng, and Xuemin Shen. 2020. PROTECT: efficient password-based threshold single-sign-on authentication for mobile users against perpetual leakage. *IEEE Transactions on Mobile Computing* 20, 6 (2020), 2297–2312.

[44] Zhiyi Zhang, Michal Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. 2021. EL PASSO: efficient and lightweight privacy-preserving single sign on. *Proceedings on Privacy Enhancing Technologies* 2021, 2 (2021), 70–87.

[45] Zhenfeng Zhang, Yuchen Wang, and Kang Yang. 2020. Strong Authentication without Temper-Resistant Hardware and Application to Federated Identities.. In *NDSS*.

[46] Lingli Zhou and Zhenfeng Zhang. 2010. Trusted channels with password-based authentication and TPM-based attestation. In *2010 International Conference on Communications and Mobile Computing*, Vol. 1. IEEE, 223–227.