CAPLets: Resource Aware, Capability-Based Access Control for IoT

Fatih Bakir, Chandra Krintz, and Rich Wolski
University of California, Santa Barbara

***Abstract*—We present CAPLets, an authorization mechanism that extends capability based security to support fine grained access control for multi-scale (sensors, edge, cloud) IoT deployments. To enable this, CAPLets uses a strong cryptographic construction to provide integrity while preserving computational efficiency for resource constrained systems. Moreover, CAPLets augments capabilities with dynamic, user defined *constraints* to describe arbitrary access control policies. We introduce an application specific, turing complete virtual machine, CapVM, alongside with eBPF and Wasm, to describe constraints. We show that CAPLets is able to express permissions and requirements at a fine grain, facilitating construction of non-trivial access control policies. We empirically evaluate the efficiency and flexibility of CAPLets abstractions using resource constrained devices and end-to-end IoT deployments, and compare it against related mechanisms in wide use today. Our empirical results show that CAPLets is an order of magnitude faster and more energy efficient than current IoT authorization systems.**

I. Introduction

The Internet-of-Things (IoT) is an integration of sensing, control, communications, and computing into ordinary physical objects in our environment. Developing secure and efficient applications for such settings is challenging because they must operate across trust domains comprising a vast diversity of computer architectures, capabilities, operating systems and resource scales. This heterogeneous and dynamically changing resource landscape requires applications to integrate simple battery-powered and wall-powered microcontrollers, more capable edge systems, and public clouds supporting hundreds to thousands of different APIs and services.

Existing IoT programming systems either focus on a specific IoT resource tier (i.e. the cloud), or attempt to repurpose and amalgamate existing tools and protocols not designed for the resource constraints, intermittent connectivity, and failure frequency of IoT deployments [1], [2], [3], [4], [5]. For example, most end-to-end IoT systems [6], [7], [8], [9], [10], [3], [11] use a publish/subscribe (pub-sub) model in which devices publish streams of data (often via a nearby broker); when supported, actuation often uses a separate protocol [12].

Similarly, for access control, many IoT systems use Transport Layer Security (TLS) [13] protocols, based on public-key cryptography [14], [10], [15]. These systems use edge devices and microcontrollers to communicate with servers and edge proxies via encrypted channels, often using RSA certificates. Although TLS addresses many security challenges, it also consumes significant resources on resource restricted devices (memory, computation, network, battery power, etc.) and depends on a number of resource-intensive operations for its security. The latter includes accurate and secure time keeping, awareness of certificate revocations, and maintenance of root certificates, among others.

Due to the implementation complexity, resource consumption, and configuration difficulty of TLS-based security, many IoT devices have weak or no access control, and are easily compromised [16], [17]. Moreover, the heterogeneity of device-maintenance interfaces and the complexities of remote device management often prevent users from updating weak or faulty implementations when improved software is available. As a result, many devices are placed behind gateways or firewalls which proxy requests when they are actually deployed [18], [19], rendering the resource expense associated with TLS-based security on the device needlessly redundant and costly.

Our goal, with this work, is to address these challenges at the device level. A full CAPLets IoT deployment uses TLS to protect client-to-client communications where clients are hosted on relatively resource-rich platforms, but removes the need for resource-restricted devices to implement or proxy a TLS connection.

The primary problem with current cloud-based approaches is that they make resource-restricted devices act as network-attached clients that access services hosted on resource-rich platforms. Because these services use technologies that support Internet web-services (e.g. in the cloud) they define security protocols that do not account for the paucity of on-device resources at the edge. As a result, devices acting as clients must use (or proxy the use of) the resource-intensive protocols mandated by the services. That is, the services define the protocols that the clients must use and, because they are often repurposed cloud-based web services, these protocols impose heavy resource loads on client devices.

Due to this model, current applications are dependent on the availability of cloud services at all times for users to access their own devices. For instance, a smart-light needs to be connected to the cloud service even when it is in close proximity to the user. This causes temporary or permananent outages when Internet connectivity is

lost [20], [21] or the manufacturer no longer supports the online service.

In this paper, we investigate the potential of inverting this relationship so that devices become *first-class* – that is, devices host services to which resource-rich clients (on cloud servers or smartphones) make requests. Such an approach requires the system software and its protocols to support client and server hosting on *any* device in an IoT deployment (from sensor to cloud). Unlike prior work, which focuses on system portability for IoT [22], [23], [24], we propose a unifying set of security protocols, called CAPLets, that complement this previous work to realize first-class devices.

To enable efficient access control in all IoT tiers including low-end microcontrollers, CAPLets replaces asymmetric digital signatures with fast symmetric MAC tags based on Macaroons [25], introduces a cheap key exchange mechanism, eliminating [1] TLS for client-server communications, and significantly reduces the storage footprint associated with authorization and certificate management. As a result, services can be hosted on sensors, on edge systems, or in a cloud. Unique to our approach is a constraint mechanism that is programmable and sufficiently flexible to represent diverse policies. As a summary of our contributions, CAPLets

- defines a capability mechanism that is sufficiently efficient for use by the least capable IoT devices (sensors, microcontrollers, battery-powered single board computers, etc.) as well as by more capable edge and cloud systems;
- defines a derivation process that augments existing mechanisms with semantic capability derivations to enable superior flexibility of controlled sharing;
- expands what a capability token can contain through the use of static and dynamic *constraints* to permit a wider range of policies to be expressed and to provide a protected metadata channel;
- uses metadata channels to offload policy implementation to trusted delegates across the deployment, e.g. to authenticate users and issue capabilities that a device can independently verify;
- defines an efficient and secure service request construction mechanism that does not depend on encrypted channels;
- defines a secure key exchange protocol by exploiting the cryptographic token construction; and
- supports efficient request validation, bootstrapping, capability sharing, and revocation.

We demonstrate the claimed efficiencies with an empirical evaluation of CAPLets that we conduct using microbenchmarks and an implementation of CAPLets for an end-to-end, first-class devices deployment, using a portable IoT operating system. Our work shows that CAPLets is an order of magnitude faster and more energy efficient compared to existing state-of-the-art IoT authorization systems.

[1] CAPLets depends on TLS for client-client communications (e.g. to transmit an identity token). However, it can replace TLS for client-server interactions in an IoT application.

II. Background and related work

Security is a challenge for end-to-end IoT systems because it requires distributed policy implementation governing a vast diversity of devices (in a multitude of trust domains) that may have limited computing, storage, network, and/or power capacities. In particular, TLS and asymmetric cryptography computations and key storage that are necessary to access most web services are costly (in terms of execution time and energy expenditure) for resource restricted devices. In addition, these public key cryptography methods can only be used for authentication in conjunction with some other mechanism for authorizing the authenticated users (e.g. Access Control Lists or Role Based Access Control). All such mechanisms require some storage capacity on the device.

Capability systems are popular for controlling access to distributed resources (e.g. web services [25], operating system processes [26], and IoT [27]) because they are inherently decentralized. Such systems use unforgeable tokens of authority to grant client access to resources and to facilitate sharing of access between clients. They can also implement the *Principle of Least Privilege* [28] (entities possess only the access rights that they actually require), which is appealing in any security context.

Capabilities describe access rights associated with a resource in a way that is forgery and tamper-proof, often using a cryptographic signature. A concrete example of a capability is a path and permission bits in a filesystem: $Capability = (\texttt{/home/alice}, \{\texttt{R,W,X}\})$. A signed capability, called a *token*, can be securely shared across a network.

To verify the token, the server for a protected resource, upon receiving a request carrying the token (which the server generated previously and signed with a secret key known only to the server), regenerates the signature from the capability body and compares it against the token signature. If they match, the server "believes" the access rights carried in the capability and executes the request on the resource if the rights permit the access.

Capabilities can support privilege reductions, termed *derivations*, without intervention by, or cooperation with, the server. For example, Amoeba [26] implements a derivation mechanism that uses commutative hash functions on access-right bitmaps to selectively reduce capability rights.

Much research has examined the use of capabilities as the basis of IoT security [29], [30], [31], [32], [33], [34] however relatively few investigations have focused on the efficiencies necessary for microcontroller deployment. The authors of [31] explore the use of ECC on a 32-bit microcontroller. Our findings in this paper show that CAPLets is two to three orders of magnitude more efficient.

Although focused primarily on cloud systems, Macaroons [25] provides an efficient distributed authorization

mechanism using cryptographically secure tokens. Both Macaroons and CAPLets use HMAC based tags for efficient generation and verification of tokens. Macaroons introduces *caveats* to contextually attenuate (i.e. constrain) tokens. CAPLets takes this design as its inspiration for its *constraints*. Macaroons, however, does not run on resource constrained devices and requires key exchange and management (for third party caveats) to enable a service to acquire proof of authorization from another service (which CAPLets avoids). CAPLets is a significant advance over Macaroons in that it

- provides a flexible, dynamic constraint mechanism (Macaroons policies are specified at deployment and do not change),
- accomodates arbitrary metadata channels,
- introduces frames to facilitate analysis and caching,
- has a more expressive, capability based construction,
- introduces a unified request mechanism,
- provides authenticated key exchange based on key derivation from token tags for efficient encryption,
- has a zero copy serialization protocol, and it
- runs on the least capable of edge devices.

Vanadium [35] also uses secure tokens and inherits the caveat design of [25]. Unlike CAPLets, however, it is based on digital certificates and ECDSA signatures. Our work shows that such use introduces significantly higher overhead. {WAVE} [36] implements distributed authorization for IoT. However, it heavily depends on public key crypto and a distributed persistent storage (ULDM), making it ill-suited for resource restricted devices. Active certificates [37] presents a novel method for distributed delegation by including code in offline tokens. However, the implementation uses Public Key Cryptography and Java, making it ill-suited to resource-restricted, device-level deployment.

Other relevant work [18], [19] studies secure gateways as a way of protecting resource restricted devices. CAPLets is compatible with and complementary to these approaches.

Cloud and web-based platforms [38], [39], [40] have yet to address resource-restriction to enable device-level deployment. An alternative system model, based on triggered computation implementing a distributed event system, is emerging as one that can support a portable, reactive, robust, and power-efficient operating system. IFTTT [41] is a proprietary commercial example while CSPOT [24] and CloudPath [42] are academic works, and CSPOT is open source. Because CSPOT also targets resource-restricted microcontrollers, we were able modify it to replace it's minimal authorization logic with CAPLets.

III. Threat Model and Assumptions

CAPLets makes specific (and somewhat common) assumptions about the network and physical security of the devices. We list them here to set a common frame of reference for the rest of the paper.

- Secrets held on devices cannot be remotely compromised.
- Recovering the Message Authentication Code (MAC) secret from a tag and a plaintext is infeasible.
- The randomness used for generating cryptographic secrets is not guessable by external attackers.
- An attacker may control every part of the network, including delaying, repeating, dropping, inspecting and modifying packets at will.

IV. CAPLets

In this section, we describe core CAPLets abstractions. A **capability**, denoted $C_{Type}(data)$ or just C when their content is irrelevant, is a typed object expressing a privilege. For instance, $C_{Dir}(read, /var/log)$ specifies "read access to /var/log" for the directory type. A **frame**, denoted F, is a set of capabilities, which carry rights to multiple objects as a single unit. A **token**, denoted T, is the unit of communication. It has a body, denoted $body(T)$, and a tag over that body. The tag is computed with a Message Authentication Code (MAC) function.

CAPLets generates tags using a Hashed Message Authentication Code (HMAC) [43], in particular HMAC-SHA256. We use HMACs because of their efficiency characteristics (we compare their performance against alternative approaches in Section V). With this construction it is safe to transmit tokens back and forth over a network as any modification to the body of a token is detectable, i.e. tokens are unforgeable.

For each device, called an origin, we define a special token R called the root token. R only contains a special frame F_R, called the root frame and F_R in turn only has a special capability, C_R called the root capability. C_R grants absolute privilege to the device it is associated with.

The tag of the root token, $tag(R)$, is computed as $MAC(Secret, F_R)$. The $Secret$ is a random secret generated on the device during bootstrapping (described below). Currently, we use hardware with true random number generators to generate this secret. Tag generation for other tokens is also described below.

Tokens do not carry information regarding their origin device. This means the holder of a token must know (or discover through some external mechanism) the server that originally generated the token, and can process the token to grant access to protected resources to the bearer carried therein. While traditional capability implementations include the responsibility of absolutely naming objects [26], CAPLets refrains from doing so. The reasons are two fold: (i) including the *name* (a DNS name, IP address or a more complicated naming scheme) of the server bloats the token sizes and (ii) in a distributed network with several, often incompatible subnetworks, naming exact servers is not a solved problem. For instance, a global address for a server behind a NAT or in a non-IP network such as Zigbee or Bluetooth does not exist.

Figure 1 shows an example of a CAPLets *root token*. The tag (top bar) is generated from the frame body, using

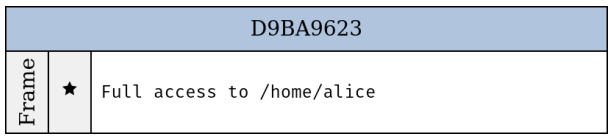

Fig. 1. The root token for `/home/alice` directory. The bar shows the token tag, which is computed as $MAC(secret, FrameBody)$. The $\star$ denotes the capabilities in a frame.

a secret known only to the resource owner (e.g. a secret stored on a device).

A. Authenticated modification

A token protects its body with the invariant that $MAC(Secret, body(T)) = tag(T)$, that is, the tag carried within the token is equal to the computed tag. While tokens with this exact form are used in practice (for instance as JSON Web Tokens [44]), they preclude the ability to perform offline policy delegations.

Past work on Macaroons [25] showed, however, that it is possible to allow some controlled modification with this cryptographic construction in the form of appending to a token's body. Basically, a token starts empty with a nonce tag always available to the origin. Anyone can append new, delimited data to the token and update its tag with another MAC. Upon receiving an appended token, an origin can confirm the integrity of the whole by replaying the modifications, starting from the empty token. Upon each append, the new tag is computed as $tag(tail :: head) = MAC(tag(tail), head)$. :: is the list append operator. Note that this operation does not depend on the secret held by the server and any party can append data to a token without any involvement from the server.

In other words, every element in the token protects the next one's integrity. This mechanism is similar to certificate verification in TLS with which a party can verify a certificate's integrity by following the signature chain starting from a Certificate Authority (CA). The difference here is that because only the server holds the origin secret, it is the only entity that can verify an entire chain.

Note that this construction is oblivious to the contents of the body. Macaroons [25] uses it to append caveats to tokens. CAPLets uses this approach but extends it to define and enforce semantic requirements associated with each append.

B. Token derivation

Specifically, CAPLets constructs token bodies as a list of frames, starting with F_R. Only the last frame of a token is used to describe its privileges. In other words, the rights granted by a token are the capabilities in its last frame. The last frame is called the *leaf frame*. The other frames are present in the token for the purposes of integrity verification and derivation checking and comprise the *derivation chain*. Formally, $body(T) = [F_R, ..., F_{leaf}]$ where $[F_R, ...]$ is the derivation chain.

We define a valid derivation to be one that reduces privileges monotonically. For instance, C_1 = "Read/write $Sensor_1$" is more privileged than C_2 = "Read $Sensor_1$". A derivation from a frame with C_2 to a frame with C_1 is deemed invalid, whereas the other way is valid. Note that this decision is application dependent and an application may define the C_1 to C_2 to be invalid as well.

A token with invalid derivations can pass the integrity check described above. Therefore, after integrity verification, CAPLets also checks that each derivation is valid in a token.

The type of a capability is used for 2 purposes: serialization and validation checking. The first is rather trivial: the type of a capability is transferred as a header in messages and during deserialization, the header is used to reconstruct the correct object. The second is the basis of derivation validation.

Whether a derivation is valid or not is a function of two frames: $ValidDerivation(F_{old}, F_{new})$. A token is valid if all adjacent pairs in its body pass the validity check.

$$\begin{aligned} &ValidDerivation(F_{old}, F_{new}) = \\ &\forall C_{new} \in F_{new}\ \exists C_{old} \in F_{old}\ C_{new} \subseteq C_{old} \end{aligned} \tag{1}$$

In this equation, the $\subseteq$ operator decides whether the new capability is a subset of the old one. Therefore, the validity check ensures that each subsequent frame in a token is a subset of the previous one.

Whether a capability is a subset of another is an application defined relation between the types of the given capabilities with the special case $\forall CC \subseteq C_R$. If the relation between two capability types is not defined, it defaults to false. The subset is a binary relation between capabilities that forms a partial order. An example of the subset relationship occurs between a directory capability $C_D(DirPath)$ and a path capability $C_P(Path)$:

$$\begin{aligned} &C_P(Path) \subseteq C_D(DirPath) = \\ &StartsWith(Path, DirPath) \end{aligned} \tag{2}$$

This particular relationship allows the users to legally derive access rights to files in directories to which they have access. For instance, if Alice holds $C_D(/home/alice)$, she can legally derive $C_P(/home/alice/hello.txt)$.

This construction allows CAPLets applications to express flexible, natural derivations. It must be noted that the party doing the derivation checking has the authority to determine what is valid or invalid. However, sharing these relationships among the server and the clients is still beneficial to minimize non-malicious, invalid derivations.

The derivations of a token is the set of all possible tokens that can be transitively derived from it and is denoted as $\mathcal{S}_T$ (for successors of T). Conversely, the tokens that can transitively derive T is denoted $\mathcal{P}_T$ (for predecessors of T). From our previous definitions it follows that $\forall TR \in \mathcal{P}_T$.

Due to the cryptographic construction, only the party who holds $T \in \mathcal{P}_D$ can compute $tag(D)$ given only $body(D)$ by taking $tag(T)$ and replaying the derivations from T's leaf to D's leaf. This property is core to how a server verifies the tag of a derived token. Upon receiving a token T, the origin replays all frames in it over R

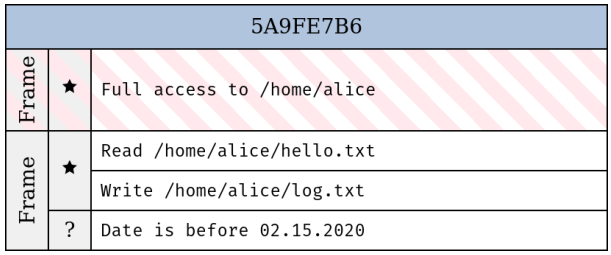

Fig. 2. A derived CAPLets token. The blue bar displays the frame tag. The root frame, now in red stripes, is still present in the token, but its contents are only used for validation purposes. ⋆'d lines denote capabilities and lines with ? denote constraints.

and checks whether the tag supplied in T matches the computed tag. If not, it rejects the token. After the tag verification, the server checks derivation validity (and rejects the token if invalid).

By maintaining the chain of derivations, tokens also carry within them a form of sharing history. That is, by looking at a token, a resource owner can trace how the holder of this token acquired it. As we explain below, combined with the use of identity tokens, this history allows for a powerful yet simple auditing, debugging, and revocation mechanism.

Figure 2 shows a derived token. The root token is at the top (the stripes showing that while it's contents are visible, they are not usable). The derived frame in the figure has two new capabilities and lacks the original one, demonstrating the subset relationship.

CAPLets' distinction between capabilities and constraints allows for efficient caching and analysis of permissions a token carries. For instance, a client can submit a token to be used in a session, which the server could verify ahead of time as much as it can (for instance time dependent constraints cannot be verified ahead of time, but endpoint constraints can). After that point, the client can efficiently perform multiple requests with that token.

CAPLets uses a zero-copy network format similar to [45], [46], improving overall verification performance.

C. Bootstrapping

To bootstrap authorization, the device generates a cryptographically random secret, which it stores in internal non-volatile memory. The internal secret is never shared externally and short of physical attacks, is unrecoverable. Using this secret with a MAC, CAPLets produces the root token R. R is then transmitted securely to the owner through the commissioning transport, typically a wired connection. For wireless commissioning, a one time secure channel is needed to transport the token. Post commissioning, the device need not create or hand out tokens (although it can).

D. Sharing

R is securely held by the resource owner, Alice. Alice uses this root token to derive privilege-reduced request capabilities for herself to use in accessing resources.

Alice can also share derivations of R with another actor, Bob. To do so, she constructs a frame with only the resources she intends to share with Bob. CAPLets appends this frame to R and computes the new tag as explained previously. She can safely pass this derived token to Bob as the MAC is one-way, i.e. even though he can see the contents R, Bob cannot recover $tag(R)$. The transfer of the token from Alice to Bob needs to occur over a secure channel (e.g. using TLS) to prevent the token from being seen by anyone other than Bob.

E. Constraints

The mechanism explained so far is limited in that an owner of a token can only restrict it by removing a permission from it. While sufficient for some purposes, sophisticated authorization policies require the ability to condition authorization on a set of circumstances that may change after a resource is instantiated.

For example, when sharing a capability for a sensor, the owner, Alice, may want to restrict the receiver, Bob, such that he can only take readings when the device has sufficient battery power. Such a restriction is difficult to express in terms of monotonically decreasing access rights. While a special case field of required battery level can be added to every token, such an approach does not scale to support a large number of restrictions.

As a generic solution for expressing arbitrary limitations on capabilities, we introduce constraints. Constraints are carried in frames alongside the capabilities and used to decide whether or not the frame is valid. That is, if a constraint is not satisfied, the frame is deemed invalid.

Constraints consist of executable code for a sandboxed virtual machine (VM). CAPLets does not specify a VM: it could be a fully generic VM like the Java Virtual Machine (JVM), or a special purpose machine like eBPF or CapVM as described in Section V. After validating a token, a server then evaluates the constraints in the leaf frame and if any constraint fails, it rejects the token.

During a derivation, an application can emit arbitrary code for the VM and add it to the leaf frame. Code contents are protected via token construction just like capabilities.

With such a construction, CAPLets deployments can be future-proof and enable the expression of truly flexible authorization policies. For instance, with Macaroons [25], the interpretation of caveats is baked in at deployment time and they are fixed and unchangeable. As a result, new policies that are needed after deployment cannot be implemented. Concretely, for the above example, if the Macaroons application did not include a *"battery level is above N%"* caveat checker at build/deploy time, it is impossible to specify such a requirement. With CAPLets, a client application can specify it dynamically.

Constraints may depend on a variety of information sources. During evaluation, a constraint has access to the contents of the tokens provided by a client and key context such as network endpoint information. They may

also access global information such as the current time or battery level through the use of VM-calls. The set of global state exposed to a constraint depends on the application. For example, it is possible to authorize the constraints of a token with other tokens for enhanced protection.

Constraints naturally integrate into the CAPLets derivation mechanism. Since ordering between arbitrary code is ill-defined, the subset relation between constraints is defined to be equality. This means that each constraint in a previous frame must appear verbatim in the next one. Since all constraints must be met for a frame to be valid, adding a new one can never escalate privileges.

Constraints may also express dependencies that span multiple tokens. A token can be constrained in a way that it is valid only if the request also presents a token with a specific signature. Or it may require that a token contains a specific capability to be valid. For example, an *identity constraint* can be used to limit a token to be only valid if it is being used by a specific user. We discuss identity implementation using CAPLets in Section IV-H.

For widely used and common constraints, CAPLets supports a static optimization where the code for a constraint can be baked into an application in an optimized form and can be delegated to by regular constraints. We call such constraints *static constraints*. For such cases, the frame need not carry any code for the constraint and just name the static constraint it depends on. Static constraints only carry the data they need, and are very efficient both in terms of network transfers and execution time, while maintaining the flexibility of CAPLets.

For example, a token can be constrained to be valid only when it is raining, as determined by reading an on-board rain sensor. Another example is an end point constraint which checks whether the request originates from a certain end point. For a static rain constraint, the body of the constraint is empty: its existence is enough to make a decision. For the end point constraint, its body must carry the specific end point to check at validation time.

F. Capability Revocation

CAPLets has two mechanisms for revocation with different guarantees: eventual and immediate. Eventual revocations are space efficient and CAPLets provides a timeout constraint for its implementation. Tokens with a timeout constraint are periodically refreshed by the issuer. An issuer may revoke a token by choosing not to refresh it at the next cycle. While this approach is good enough for many applications, CAPLets also optionally supports immediate revocations if needed, which requires storage at the server (as described below).

When a token holder wishes to immediately revoke a token she has shared, she sends a revocation request to the origin. The server places the tag of the token on a blacklist. Any request that contains the token with a tag in the blacklist is rejected by the server. The check is applied on every frame and therefore, revoking T implicitly revokes $\forall DD \in \mathcal{S}_{\mathcal{T}}$ as well. To prevent the unbounded growth of the blacklist, each entry carries the expiry date of the token along with the tag. Upon expiration, the entry is removed from the blacklist since the expiry mechanism will prevent the use of the token. In the unlikely event the list grows to unacceptable levels, the owner may revoke R, which revokes every token and clears the list.

Note that the device in this case need not refresh tokens – it only maintains the blacklist – since the tokens may be (and usually are) derived by the owner on a resource-rich platform. That is, the owner may issue (and refresh) tokens that are validatable by the device without communicating with the device.

G. Request Construction & Validation

All other capability systems use the token only to authorize a request delivered alongside the capability. The server checks if the provided capability has permissions to perform the request. Because the request is included separately, it is possible (and often convenient or expedient) to transmit a capability with higher privilege than the request requires. This is arguably a violation of the Principle of Least Privilege [47].

CAPLets takes an alternative approach and unifies service requests and capabilities. We exploit the fact CAPLets capabilities are arbitrary objects, and encode requests as special capabilities. Each service request is expressed as its own capability type by placing it in a leaf frame. For instance, a `read` request on a file can be placed inside a leaf frame where the previous frame includes read access rights for the file. In this case, the file read permission is a superset of `read` request capability on that file. CAPLets' construction ensures the derivation checks succeed only if the frame before the request frame has sufficient access rights to perform the request. We denote request capabilities as C_{Req}.

A request capability must specify an operation precisely. We enforce this by mandating a request capability not be the superset of any other capability. In practice this means that any frame containing a request capability cannot be further derived. This ensures that once a request token is crafted, it is immutable. It can be performed as it is or discarded, but partial application is not allowed. This construction ensures the integrity of requests even when transmitted in plain text, allowing for enhanced efficiency when confidentiality is not needed.

It is sometimes convenient to create requests that depend on other tokens, for instance as proof of an earlier action such as authentication. Tokens passed alongside a request token for supporting purposes are called *auxiliary tokens*. To prevent an actor from using the auxiliary tokens separately with other, unintended, requests, CAPLets requires that clients to place a *signature constraint* with the signature of the request token in the leaf frames of each auxiliary token. This constraint ensures that a token is only valid if it is being used alongside a token with the specified signature.

C8AFB66B		
Frame	★	Full access to /home/alice
Frame	★	Read /home/alice/hello.txt
		Write /home/alice/log.txt
	?	Date is before 02.15.2020
Frame	★	Perform Request write(/home/alice/log.txt, "hello")
	?	Date is before 02.15.2020
		Source IP is 169.231.10.245
		Sequence number is less than *16024*

Fig. 3. A request token comprising implied rights carried by previous frames. A request token may only have request specifications and constraints in it's last frame. Since the request is delivered as a derivation, no further checks are necessary.

Upon receiving a request, a server performs three validation steps on all tokens:

Signature verification: The server reconstructs the final tag by starting with the root tag and replaying derivations in the chain. If the resulting tag does not match the tag provided by the token, the request is rejected.

Derivation validation: Every link in the derivation chain is validated for valid derivations as described in IV-B. Every constraint in the previous frame must be verbatim present in the next frame. If these conditions are met in every link in the chain, the last frame is legally derived by definition. If any invalid derivation is performed at any step, the request is rejected.

Constraint validation: Constraints in all the leaf frames are checked. If any of the constraints are unmet, the request is rejected.

If the request is not rejected in any step, it is served.

H. Identities in CAPLets

To represent identities in CAPLets, we introduce *identity capabilities*. An identity capability for a certain identity, for instance Bob, is proof that the holder of such a capability is indeed, Bob. The issuance of such identity capabilities is not directly specified by CAPLets. They can be directly issued by the owner of the device, Alice, if she wishes to implement an authentication service herself. Identity capabilities use the same mechanism as regular capabilities, and thus support derivations. This means that the issuance of identity capabilities can be offloaded to a trusted identity provider, as with regular capabilities. This provider can use existing authentication services such as LDAP [48] and issue CAPLets identity tokens for successful authentications. Such tokens should have a timeout constraint to facilitate revocation. Since the tokens are generated by a computationally powerful and non-power-constrained machine, users can routinely refresh them.

With this formulation, the user Bob can acquire a token, T_{Bob}, that proves to the device that he is Bob. Note that regardless of how Bob receives it, $T_{Bob} \in \mathcal{S}_{\mathcal{R}}$ and it can be verified with the regular derivation process, precluding the need for a complex public key management for the server. Either the owner issued the token directly, or she shared an intermediary identity root token with an identity provider, which then derived an identity token for Bob from it.

An identity constraint limits a token to be only valid if presented alongside a specific identity capability. When Alice wishes to limit a token to be only used by Bob, she places an *identity constraint* on the token and shares it with Bob. When performing a request with that token, Bob includes his identity token, which the server checks using a constraint validator. Without the identity token, the access right token will not be validated. Figure 4 demonstrates an end to end interaction with an external identity provider.

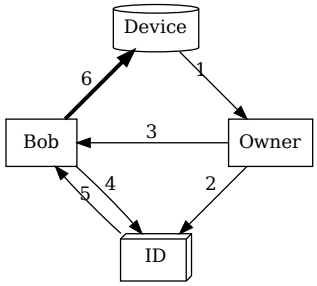

Fig. 4. 1. Device delivers the root token to the owner during provisioning. 2. The owner registers the device with an identity provider service. 3. Owner shares a Bob-id-constrained token with Bob. 4-5. Bob receives a Bob-id token to be used with the device. 6. Bob delivers a request token alongside his id token. The thick edge denotes a client-server communication and is a CAPLets channel. The thin edges use a TLS based protocol, such as HTTPS or SSH.

An identity token is only ever used as an auxiliary to another request token and is protected by the signature constraint explained in Section IV-G. Note that the body of the identity capability and constraint types are dependent on the desired identification mechanism. The possibilities range from a single integer to variable length strings encoding complex identities. Since an identity token generated for a specific server will not be valid for a different server due to signature mismatches, even a single integer provides security.

I. CAPLets Key Exchange

So far we have focused on authorization. While CAPLets can securely perform request authorization without an encrypted channel due to its immutable request construction, the bodies of requests and responses are visible to attackers. Moreover, responses are not authenticated. The conventional approach to this problem is to employ TLS channels [3], [10], [25], [35]. However, TLS based encryption has many drawbacks including certificate management on devices and, as we quantify in our evaluation, requiring slow and power hungry operations. While TLS might be as efficient as possible for the guarantees it makes, we believe that not all such guarantees are as desirable in an IoT context (versus an Internet/cloud context). In this section, we describe how we exploit the

cryptographic construction to provide a limited but efficient form of encrypted channel among CAPLets parties. We provide a formal proof of the security of our algoritm against passive, replay, and man in the middle attacks as an Appendix.

At the core of our algorithm is the observation that $tag(T)$ forms a shared secret among the users who hold any token in $T \cup P_T$. That is, a party can recover the tag from a body if and only if they hold a token in this set.

As described above, we use this property to ensure token integrity. However, it is also possible to use it to derive a secure symmetric key among anyone within this set, particularly, between the server and any user, since the server holds R, the root token.

For this purpose, we introduce a new leaf capability type: an encryption capability, denoted as E. An encryption capability E has the following properties: $\forall CE \subset C$ and $\forall CC \not\subset E$. This means that an encryption object can be derived from any capability and no other capability can be derived from an encryption object.

When a client wishes to set up an encrypted channel with anyone holding $T \cup P_T$, they derive an encryption token T_E from T that has only the encryption capability in the leaf frame and **transmits the token body without the tag** to the server. Upon receiving an encryption token, the server will recover the shared secret $tag(T_E)$. At this point, both parties can generate matching keys k_{T_E} for encryption through a Key Derivation Function (KDF). Both parties then challenge the other side to compute $MAC(tag(T_E), N_{\{C,D\}})$, $N_{\{C,D\}}$ is chosen at random by each party respectively.

The challenge uses the same construction we use to compute derived tags and may seem susceptible to an attack where an attacker asks a party to compute the tag of a S_{T_E}. For this reason, we have defined T_E to be non-deriveable, i.e. $S_{T_E} = \varnothing$. While the cryptographic construction allows such a token, the logical construction prevents its use. This is a case where past approaches to capability construction are insufficient to accomodate arbitrary metadata channels.

Instead of using $tag(T_E)$ directly to derive encryption keys, an authenticated Ephemeral Diffie-Hellman (DHE) key exchange can be employed to enhance this algorithm with Perfect Forward Secrecy (PFS). In DHE key exchanges, both parties generate a temporary public and private key pair and share the public parts over the network. Upon receiving the public key of the other party they combine their private key and the public key of the other party to agree on the same key. Since any potential attacker only sees the public parts, they cannot compute the shared key. However, unauthenticated DHE is susceptible to trivial Man in the Middle (MitM) attacks. To prevent this, the DH public keys are authenticated. For instance, if both parties had long term public keys, they could sign their messages so that the other end can confirm with the long term public key.

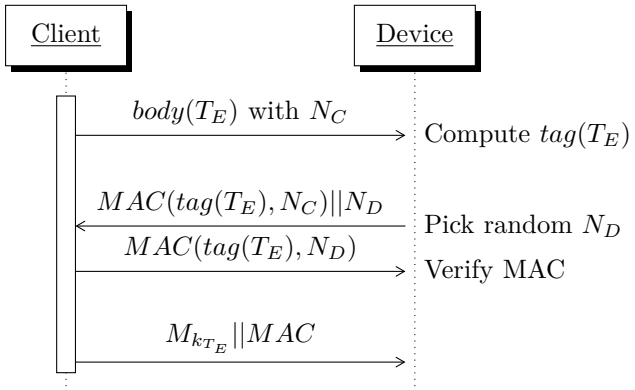

Fig. 5. CAPLets key exchange: $tag(T_E)$ is not recoverable by an attacker and forms a shared secret. Both parties can ensure the other end *knows* the secret by sending each other challenges. Once authentication is done, the session key k_{T_E} is computed as $KDF(tag(T_E))$. M_k means the encryption of M with key k with some symmetric cipher. The MAC is computed over M_k.

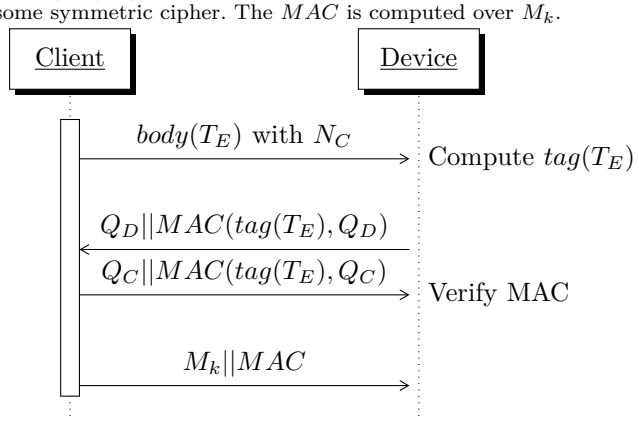

Fig. 6. Perfect-forward-secrecy secure CAPLets key exchange: $tag(T_E)$ is not recoverable by an attacker and forms a shared secret. Both parties generate ephemeral public-private key pairs $(Q_C, d_C), (Q_D, d_D)$ and share tagged public keys. Parties authenticate each other by verifying the received MAC. After authentication, session key k is computed as $d_C * Q_D$ and $d_D * Q_C$ for the client and device, respectively. Note that the session key is not derived from $tag(T_E)$. M_k is the encryption of message M with key k via some symmetric cipher. The MAC is computed over M_k.

In the case of TLS, a digital signature algorithm such as RSA or ECDSA (Elliptic Curve Digital Signature Algorithm) must be used for this purpose. However, digital signatures are compute intensive and consume significant energy. In CAPLets, the parties instead use the shared secret $tag(T_E)$ with an HMAC to authenticate their DH public keys (Q_C and Q_D in Figure 6).

For CAPLets, PFS means that if $tag(T_E)$ is compromised at some future time, an attacker cannot decrypt any past communication even if they stored all the packets between the parties. However, as we will demonstrate in our evaluation, PFS consumes an order of magnitude more energy than the rest of the entire communication. In practice, IoT communication often loses value with time, and by the time an attacker recovers a tag and decrpyts past data points, the data may already be obsolete. When PFS is necessary, the CAPLets implementation is still more effficient than TLS due to its ability to use MACs instead of digital signatures.

	CC3220SF	nRF52840	ESP8266
Processor	80 MHz Cortex-M4	64 MHz Cortex-M4	80 MHz Xtensa
Memory	256KB	256KB	80KB
Crypt HW	Yes	Yes	No
Network	WiFi	BLE	WiFi
Coremark	87	212	191

TABLE I
32-BIT DEVICES USED IN THE EXPERIMENTS.

Once the key exchange is finished, the application is free to use the key with any suitable symmetric cipher. Our prototype uses AES-CTR and HMAC-SHA256 in a correct encrypt-then-MAC construction.

As mentioned above, anyone holding a token in P_{T_E} can actively intercept this key exchange. Since $tag(T_E)$ is the only shared knowledge among the device and a user, this is impossible to prevent (even with TLS, certificates would have to be pre-exchanged, i.e. another shared knowledge). However, $|P_{T_E}|$ can be minimized, for instance by the owner issuing special purpose tokens to users that are only used to derive encryption tokens. In that case, $|P_{T_E}| = 1$, with the device owner being the only other party who can listen to the conversations. We view this last point as a desirable property as a device owner should be able to monitor communication to/from the devices they own.

V. Experimental Evaluation

In this section, we evaluate CAPLets using an end-to-end experiment that couples an IoT device at the edge with the cloud. We compare CAPLets against two mature and commercially available systems for implementing IoT applications using cloud computing. We also analyze the performance of CAPLets using a set of microbenchmarks to help illuminate its efficiencies.

A. Devices, Software, and Setup

Table I shows the resource constrained hardware platforms that we consider as end devices in this study. We consider three 32-bit microcontrollers, each with different resource configurations (processor speed, memory size, and network type). Two of the three devices have hardware that performs cryptographic operations efficiently (demarked `Crypt HW` in the table). Coremark [49] refers to the approximate compute power of each processor.

We execute CAPLets on these devices on bare metal, using cloud SDKs and FreeRTOS from Amazon Web Services (AWS) and Microsoft Azure, and using CSPOT [24] – an operating system and runtime designed for cloud and IoT applications. For all experiments, we optimize for code size and use the same drivers across devices.

Our bare metal environment consists of a minimal implementation designed to achieve and measure the maximum efficiency of the methodology in isolation. As Macaroons [25] has no implementation that can run on a microcontroller, we benchmarked it and CAPLets on an x86 PC.

We conduct experiments using two deployments:

Edge: The device communicates with an edge system (i.e. an Intel NUC [50]) on the same WLAN network. For AWS and Azure experiments, the device communicates with a Greengrass or IoT Edge virtual machine instance on the NUC over MQTT [6]. CAPLets communicates with the instance via TCP.

Cloud: The device communicates with resources hosted in a public cloud (connected via a fast academic network located in California). For AWS and Azure, the device communicates with an IoT Core or IoT Hub instance, respectively, over MQTT. These instances are located in Oregon and California, respectively. CAPLets uses an AWS EC2 instance in Oregon.

We configure AWS and Azure software (and reprogram the device) to use the different communication end points (edge and cloud). For CAPLets, neither the software on the device nor on the edge had to be modified since the device is oblivious to the origin and the same authorization mechanism works regardless of the client location (i.e. it uses our first-class-devices model).

We measure energy consumption by sampling the momentary power use of the processors. We use an INA219 sensor which we set to 2KHz (the maximum sampling rate the sensor supports) and the highest resolution.

B. End-to-end Evaluation

As an end-to-end experiment, we measure the time and energy costs of a sensor capturing and communicating a sample to a user. We implement this application using the services and software provided by the leading IoT service providers, AWS and Azure.

Specifically, we use Amazon FreeRTOS on the device side, AWS Greengrass on the edge, and AWS IoT Core on the cloud for AWS. For Azure, we use FreeRTOS with the Azure SDK, IoT Edge, and IoT Hub for device, edge, and cloud, respectively. Communication is handled by AWS or Azure IoT SDK libraries, which use an MQTT [6] implementation of a publish-subscribe protocol between the end device and edge or cloud. The providers issue each device a private key and certificate to communicate with the services. We inject the key into our CC3220SF device at firmware build time. When possible, CAPLets and the SDKs perform the cryptographic operations in hardware.

We also evaluate an implementation of the application for CSPOT [24] – an experimental and open source operating system designed specifically for coupled IoT-and-cloud applications. CSPOT is useful to this study because it is a complete system capable of implementing device-level services directly, without a proxy, as well as services hosted by edge and cloud resources.

AWS and Azure implement proxy-based approaches in which an edge device or a cloud resource (but not a resource-restricted device such as a microcontroller) store access control lists for all users and devices. The resource also performs authentication.

In AWS and Azure, the device wakes up every 5 minutes, collects a sample, associates with the WiFi network, es-

Operation	Time ms	Enrgy mJ	Code KB
No Auth Edge	95 (20)	21 (4)	45
CAPLets Edge (bare metal)	99 (28)	22 (6)	48
No Auth Edge + CSPOT	119 (12)	22 (2)	49
CAPLets Edge + CSPOT	119 (16)	24 (3)	51
AWS Greengrass	825 (231)	148 (49)	75
Azure IoT Edge	1715 (119)	251 (22)	75
CAPLets Cloud	121 (28)	26 (5)	48
CAPLets Cloud+CSPOT	165 (25)	32 (6)	51
AWS IoT Core	1696 (908)	314 (90)	230
Azure IoT Hub	3168 (244)	457 (32)	130

TABLE II
End-to-end application performance. All execution measurements and standard deviations (in parentheses) are over 100 consecutive executions. We consider an edge (top table) and cloud deployment (bottom table).

tablishes a secure connection to the respective server over TLS (authorization is implicit in the establishment of this channel), sends the sample over MQTT and goes back to deep sleep. When users wishes to view the samples, they visit the cloud/edge service and read the samples.

For CAPLets, the device wakes up every 5 minutes, collects a sample, records it and goes back to deep sleep. When a user wishes to view the samples, they initiate a connection to the device. As we employ 802.11 Long Sleep Interval (LSI) feature, the device is soon woken up by the network processor, the communication succeeds, the user authorizes herself to the device, and receives the samples. This means that for *uninteresting samples*, i.e. samples that the user never reads, CAPLets conserves battery power. LSI allows a WiFi station to stay associated with the access point during prolonged sleeps. Instead of dropping received packets, the access point buffers them until the next beacon frame sent out frequently (at least once every few seconds). The beacon frames are handled by the low level network hardware and the device does not wake up unless there is a packet for it. Therefore, a CAPLets user will not wait for 5 minutes before they can read existing samples, but at most a few seconds.

We measure end-to-end performance and energy use from the sensor server (end device) perspective. We execute the application 100 times for each deployment (Azure, AWS, CAPLets, and CAPLets with CSPOT using edge and cloud configurations). We compute the average and standard deviation for awake time in milliseconds and energy use in millijoules. Awake time is the time it takes for the processor to wake from deep sleep and associate with WiFi network to handle one sensor sample and go back to sleep. We also measure application code size. Table II shows the results.

We include `No Auth` experiments for completeness which is CAPLets request handling with security checks disabled. These results show that the CAPLets security checks only account for about 4% of the overall processing time as can be seen by comparing the No Auth row with CAPLets Edge. CAPLets code size with or without CSPOT is 4 and 2 times smaller than that of Amazon FreeRTOS with both AWS and Azure SDKs, making space for more features on the end device. The results for CSPOT show that CAPLets, when integrated with a full system that spans device, edge, and cloud tiers in an IoT deployment, adds no discernable overhead.

The latency improvement should not be considered only in the context of user-perceived latency, but also in the context of battery life. As the CAPLets application serves the request and goes back to deep sleep, TLS based versions are still busy performing the handshake. As this operation is performed for every sample for the cloud systems, it inevitably will consume its battery earlier, resulting in unavailability.

For the cloud deployment, AWS and Azure both require the devices to be awake for an order of magnitude longer per request than using CAPLets. Part of the disparity stems from the fact that our approach minimizes the responsibilities the end device has to perform per sample. Consider the steps taken *for each data point* on the AWS set up: (i) perform a DNS look up to determine the remote end point in AWS, (ii) perform time synchronization for certificate expiry verification (iii) verify the authenticity of the remote end point by walking a certificate chain (iv) establish a TLS session, (v) establish an MQTT session, and (vi) transmit the sample.

Although some steps can be cached, and *are in these experiments*, they must execute periodically on the end device. Except for MQTT, no protocol involved in these steps was designed for a memory, processor, and power constrained microcontroller. Our approach offloads costly operations to the clients. For instance, instead of the device performing a DNS lookup to locate the server, the client performs a DNS lookup to locate the device.

C. Microbenchmark Evaluation

In this section, we conduct a broad range of microbenchmark experiments to expose the performance characteristics of CAPLets. In particular, we evaluate the costs of cryptographic primitives that are performed per request for CAPLets versus competitive approaches.

In its core CAPLets depends on 2 cryptographic primitives: HMAC-SHA256 for token construction and message authentication and AES-CTR for securing the communication. If PFS key exchange is used, it also uses ECDHE for temporary session key exchange.

The primary competitors of CAPLets are TLS-based cloud services and Vanadium [35]. Both these approaches use asymmetric cryptography to authenticate parties. For

Operation	CC3220SF	nRF52840	ESP8266
ECDHE (SW)	13679.07	1596.87	116592.71
ECDHE (HW)	N/A	537.45	N/A
AES128-CTR (SW)	73.95	9.76	96.72
AES128-CTR (HW)	19.70	1.73	N/A
HMAC-SHA256 (SW)	75.92	8.42	93.32
HMAC-SHA256 (HW)	10.65	4.94	N/A
ECDSA Sign (SW)	15592.68	2730.87	122617.24
ECDSA Sign (HW)	15337.49	456.05	N/A
ECDSA Verify (SW)	16914.24	3007.92	138659.75
ECDSA Verify (HW)	17023.39	457.65	N/A

TABLE III
AVERAGE ENERGY CONSUMPTION FOR MICROBENCHMARKS OVER 100 RUNS. THE UNITS ARE MICROJOULES.

Handshake	Energy Use
TLS PFS	1451.16
CAPLETS PFS	547.33
CAPLETS Non-PFS	9.88

TABLE IV
ENERGY CONSUMPTION FOR THE HANDSHAKES OF TLS AND CAPLETS ON NRF52840. THE UNITS ARE MICROJOULES.

IoT systems, Elliptic Curves are preferred [51], [52] due to their smaller key sizes for equivalent security [53] and better performance characteristics when compared to RSA [54]. Therefore, we only consider cipher suites with ECDSA.

For the ECDHE experiments, we use Curve25519 for its efficient implementation [55]. For the ECDSA experiments, we use the p256 [56] curve. These experiments demonstrate the cost of using long term public keys for parties to authenticate themselves. These operations are performed when a device acts as a client for cloud or edge service, as in AWS IoT Core and Greengrass.

Table III presents the average energy consumption in microjoules for each microbenchmark for the three hardware platforms. Note that timings for this set of microbenchmarks are comparatively similar showing the same ratios of performance in comparison (i.e. CAPLETS is one to two orders of magnitude faster). We omit the timing comparison due to space constraints.

TLS based protocols sign ECDHE messages with ECDSA in the handshake, so their cost is ECDHE + ECDSA Sign + ECDSA Verify. With CAPLETS, the cost is ECDHE + 2 * HMACs. Our results show that an HMAC uses 2 to 3 orders of magnitude less energy than either ECDSA operation. After key exchange, both TLS and CAPLETS switch to an efficient, symmetric cipher.

Our efficiency gains stem from the fact that over a connection, a TLS protocol uses every operation in Table III. CAPLETS on the other hand eliminates the use of the bottom cluster, and uses ECDHE only for PFS. Table IV shows that with PFS, a CAPLETS handshake consumes about 3 times less energy compared to TLS. If PFS is not needed, CAPLETS consumes at least 2 orders of magnitude less energy.

Note that the hardware-based ECDSA implementation

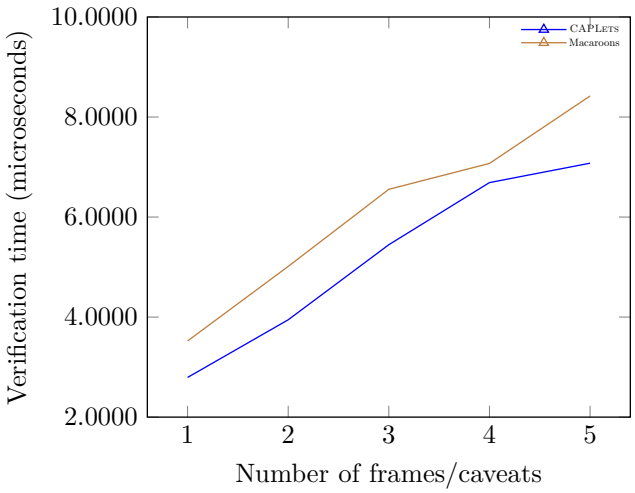

Fig. 7. Time it takes for CAPLETS and Macaroons to verify a token.

on the CC3220SF performs on-par with or worse than the software implementation. The ECDSA "acceleration" takes place on the network processor of the CC3220SF, which has a binary-blob firmware, preventing us from investigating further. This observation, however, is consistent with existing literature [57]. Thus we consider it anomalous and likely due to a firmware bug or misconfiguration specific to ECDSA on the CC3220SF.

Finally, we benchmark CAPLETS and Macaroons verification operations for different frame counts (using frames in CAPLETS and and caveats in Macaroons). We show the results in Figure 7. Surprisingly, CAPLETS performs slightly but consistently better than Macaroons even though the cryptographic construction is the same. To ensure there is no difference in cryptographic implementation, we modified our code to use the HMAC implementation used by Macaroons. We find that the performance difference is due to the use of zero copy deserialization in CAPLETS. Once Macaroons receives a packet over the network, it allocates memory for each caveat and deserializes its format into the buffers, which take linear time. CAPLETS, alternatively, operates on the received buffer directly with no deserialization step.

D. Evaluating Constraints

We next evaluate the performance of dynamic constraints (cf Section IV-E). We consider three virtual machines: (i) eBPF [58], which is used in the Linux kernel for policy control; (ii) a popular VM called WebAssembly [59] (Wasm); and a low-level VM that we developed, called `CapVM`, that is specifically designed and optimized for CAPLETS.

eBPF is a VM originally designed to filter network packets without involving the user space. Since the programs run in the kernel space, the runtime is well-isolated and its programs are limited in many ways (execution limits, 5 parameter limit for functions, lack of a linker etc).

Wasm is a general purpose VM designed for efficient execution in web browsers. It defines a stack machine and a memory-safe execution environment so that "unsafe" languages such as C or C++ can use it as a compilation target for execution in a browser. In this study, we use state-of-the-art eBPF and Wasm interpreters [60] and [61].

Our goal with the CapVM bytecode design is compactness and direct interpretability on embedded systems. Our design uses a register based ISA and provides complex instructions to access system resources and validation context, in addition to those for arithmetic, logical, and control operations. Its bytecode uses variable size instructions for maximum code compactness with configurable fixed size opcodes for efficient decoding. The ISA is untyped and does not perform any dynamic type or memory safety checks. For sandboxing, all the memory accesses are confined to the fixed, linear memory of the VM, so any unsafe user code is contained within the program and cannot affect the rest of the system. Any fault during the execution results in the current request being rejected. CapVM will be released as open source if/when this paper is published.

To evaluate the performance of this bytecode and its interpretation, we use the CC3220SF device. We consider a simple timeout constraint for this study. The constraint, written in C++, renders a token invalid after a certain amount of time has passed. We compile the code to eBPF and Wasm using clang [62] with size optimizations. Because no higher level language yet exists that can emit code for CapVM, we write the constraint code by hand, directly in the CapVM bytecode language. We impose an executed instructions limit to all VMs to prevent policies from running indefinitely. In case an execution exceeds the limit, the constraint will be rejected.

We show the measurements from these two implementations of the bytecode constraint in Table V-D. We evaluate VM Code Size (row 1), Bytecode Size (row 2), Validation Memory size (row 3), and Validation Time (row 4). Columns two to four show the results for the various alternatives. The final column (for reference) shows the metrics when we implement the constraint in native code.

This experiment shows some interesing results. CapVM is an order of magnitude faster and consumes around 1% the energy of Wasm. eBPF, on the other hand, has similar performance and energy use characteristics. However, its bytecode is large compared to CapVM, and double that of Wasm. Code sizes (of their interpreters) of CapVM and eBPF are similar and add between 1 and 7 kB to the static version. Wasm, on the other hand, takes up about 55 kB of space due to its optimization pipeline. Our results show that this VM's focus on high performance hurts its memory use and performance significantly. Since policies are one off executions rather than long running tasks, we believe Wasm is inappropriate for such use.

The large bytecode difference between CapVM and eBPF can be explained by eBPF still being a general purpose VM while CapVM is designed specifically for the interpretation of constraint checkers in resource constrained devices. For instance, CapVM provides instructions to navigate a token, whereas eBPF must emit code to do the same. Finally, eBPF uses a fixed, 8 byte instruction encoding, bloating programs.

	CapVM	eBPF	Wasm	Static
Code Size	10.5 kB	16 kB	65 kB	9.5 kB
Bytecode	23 Bytes	400 Bytes	193 Bytes	N/A
Memory	216 Bytes	512 Bytes	30510 Bytes	16 Bytes
Time	144 μs	248 μs	10560 μs	11 μs
Energy	16 μJ	20 μJ	1125 μJ	2.5 μJ

TABLE V
PERFORMANCE OF BYTECODE CONSTRAINT IMPLEMENTATIONS ON VALIDATION OPERATION: CAPVM, EBPF AND WASM. CODE SIZE IS FOR EACH VM IMPLEMENTATION WITH THE CONSTRAINT; BYTECODE SIZE IS THE SIZE OF THE RENDERED CONSTRAINT. STATIC SHOWS THE SIZE OF A DIRECT C++ COMPILATION OF THE CONSTRAINT.

In conclusion, both eBPF and CapVM show sufficient performance, energy and memory use characteristics to implement CAPLETS constraints. Having to generate CapVM assembly manually is tedious, and is, at present, a disadvantage. However, the smaller byte code size may warrant the inconvenience. Further, eBPF imposes some strict restrictions: functions can have up to 5 arguments, any more and the function does not compile. There also is no standard linker for it, so programs must be written in a single translation unit. Wasm has no such limitations, but its state-of-the art interpreter is large and less performant in terms of speed, energy, and size.

VI. CONCLUSION

In this paper, we present CAPLETS, an efficient, secure, and flexible distributed access control mechanism that provides a uniform authorization model for a broad spectrum of resource scales and non-trivial policies. CAPLETS is uniquely optimized for IoT deployments with resource restricted devices, such as microcontrollers, as the main actors. In this setting, the optimizations it provides are substantial. We believe this consideration is key to facilitating greater and more efficient security mechanisms for IoT settings.

We design CAPLETS to execute efficiently on the least-capable devices in these deployments (i.e. microcontrollers/sensors with little memory and processing power, batteries, and duty cycles), while being able to scale up for use on the most capable (cloud systems). We also define new abstractions for CAPLETS that can be used to build a wide range of efficient security measures for common attack scenarios and to facilitate encrypted channels. We empirically evaluate the performance of CAPLETS and compare it against state-of-the-art authorization mechanisms in use today. We find that CAPLETS is an order of magnitude faster and more energy efficient than this prior work for both resource constrained devices and end-to-end IoT (i.e. sensor-edge-cloud) deployments. We also show that CAPLETS performs similarly or better than

Macaroons [25], while qualitatively improving portability and flexibility and introducing new features, including secure key exchange without the use of TLS.

References

[1] "AWS Lambda IoT Reference Architecture," http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html [Online; accessed 1-Nov-2016].

[2] Microsoft, "Iot edge: Microsoft azure." [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/

[3] "GreenGrass and IoT Core - Amazon Web Services," https://aws.amazon.com/iot-core,greengrass/, [Online; accessed 2-Mar-2019].

[4] Microsoft, "Iot hub: Microsoft azure." [Online]. Available: https://azure.microsoft.com/en-us/services/iot-hub/

[5] N. Berdy, "How to use Azure Functions with IoT Hub message routing," 2017, "https://azure.microsoft.com/en-us/blog/how-to-use-azure-functions-with-iot-hub-message-routing/".

[6] A. Banks and R. Gupta, "Mqtt v3.1.1 protocol specification," 2014.

[7] A. Stanford-Clark and H. Truong, "Mqtt for sensor networks (mqtt-sn) protocol specification," 2013.

[8] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, "Wheres the bear?–automating wildlife image processing using iot and edge cloud systems," in *ACM Conference on IoT Design and Implementation*, 2017.

[9] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.

[10] "Azure Internet of Things," https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite, [Online; accessed 22-Aug-2016].

[11] "Fog Data Services - Cisco," http://www.cisco.com/c/en/us/products/cloud-systems-management/fog-data-services/index.html, [Online; accessed 22-Aug-2016].

[12] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Network Applications*, vol. 24, 2019.

[13] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Internet Requests for Comments, IETF, RFC 8446, August 2018.

[14] "Internet of Things - Amazon Web Services," https://aws.amazon.com/iot/, [Online; accessed 22-Aug-2016].

[15] "Internet of Things Solutions - Google Cloud Platform," https://cloud.google.com/solutions/iot/, [Online; accessed 22-Aug-2016].

[16] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *USENIX Security Symposium*, Aug. 2017.

[17] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, "Security vulnerabilities of internet of things: A case study of the smart plug system," *IEEE Internet of Things Journal*, vol. 4, no. 6, Dec 2017.

[18] R. Ko and J. Mickens, "Deadbolt: Securing iot deployments," in *Proceedings of the Applied Networking Research Workshop on*, 2018, pp. 50–57.

[19] R. Trimananda, A. Younis, B. Wang, B. Xu, B. Demsky, and G. Xu, "Vigilia: Securing smart home edge computing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 74–89.

[20] M. Bergen, "Google outage reignites worries about smart home without backups." [Online]. Available: https://www.bloomberg.com/news/newsletters/2020-12-16/google-outage-reignites-worries-about-smart-home-without-backups

[21] iRobot, "An amazon aws outage is currently impacting our irobot home app..." [Online]. Available: https://twitter.com/iRobot/status/1331667670383685635

[22] X. Shelby, K. Hartke, and C. Borman, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, 2014.

[23] V. Eswara, G. Srivastava, and S. Biswas, "Riotnet: Reactive iot control network," in *IEEE International Conference on Internet of Things*, June 2017.

[24] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "Cspot: Portable, multi-scale functions-as-a-service for iot," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 236–249. [Online]. Available: https://doi.org/10.1145/3318216.3363314

[25] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Network and Distributed System Security Symposium*, 2014.

[26] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba – A distributed Operating System for the 1990's," *IEEE Computer*, vol. 23, no. 5, May 1990.

[27] S. Gusmeroli, S. Piccione, and D. Rotondi, "A Capability-based Security Approach to Manage Access Control in the Internet of Things," *Mathematical and Computer Modelling*, vol. 58, no. 5-6, 2013.

[28] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of The ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[29] S. Gusmeroli, S. Piccione, and D. Rotondi, "A capability-based security approach to manage access control in the internet of things," *Mathematical and Computer Modelling*, vol. 58, no. 58, pp. 1189–1205, 2013.

[30] P. N. Mahalle, B. Anggorojati, N. R. Prasad, and R. Prasad, "Identity authentication and capability based access control (iacac) for the internet of things," *Journal of Cyber Security and Mobility*, vol. 1, no. 4, pp. 309–348, 2012.

[31] J. L. Hernandez-Ramos, A. J. Jara, L. Marin, and A. F. S. Gomez, "Dcapbac: embedding authorization logic into smart things through ecc optimizations," *International Journal of Computer Mathematics*, vol. 93, no. 2, pp. 345–366, 2016.

[32] S. Gusmeroli, S. Piccione, and D. Rotondi, "Iot access control issues: A capability based approach," in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2012, pp. 787–792.

[33] R. Xu, Y. Chen, E. Blasch, and G. Chen, "A federated capability-based access control mechanism for internet of things (iots)," in *Sensors and Systems for Space Applications XI*, vol. 10641, 2018.

[34] ——, "Blendcac: A blockchain-enabled decentralized capability-based access control for iots," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1027–1034.

[35] A. Erbsen, A. Shankar, and A. Taly, "Distributed authorization in vanadium," *arXiv preprint arXiv:1607.02192*, 2016.

[36] M. P. Andersen, S. Kumar, M. AbdelBaky, G. Fierro, J. Kolb, H.-S. Kim, D. E. Culler, and R. A. Popa, "WAVE: A decentralized authorization framework with transitive delegation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1375–1392.

[37] N. Borisov and E. A. Brewer, "Active certificates: A framework for delegation," in *NDSS*, 2002.

[38] S. Authors, "Spring framework." [Online]. Available: https://spring.io/projects/spring-framework

[39] N. Authors, "Node.js web application framework." [Online]. Available: https://expressjs.com/

[40] "Google app engine," "http://code.google.com/appengine/".

[41] I. Authors, "Ifttt, if this then that," accessed 22-May-2020. [Online]. Available: https://ifttt.com/

[42] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara, "Cloudpath: a multi-tier cloud computing framework," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, p. 20.

[43] H. Krawczyk, R. Canetti, and M. Bellare, "Hmac: Keyed-hashing for message authentication," 1997, [Online; accessed 26-Apr-2019] https://tools.ietf.org/html/rfc2104.

[44] "Json web token (jwt)." [Online]. Available: https://tools.ietf.org/html/rfc7519
[45] Google, "Flatbuffers," https://google.github.io/flatbuffers/, 2021.
[46] Sandstorm.io, "Cap'n proto," https://capnproto.org/, 2021.
[47] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of The IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.
[48] "OpenLDAP," "http://www.openldap.org/".
[49] EEMBC, "Coremark, an eembc benchmark," https://www.eembc.org/coremark/, 2021.
[50] "Intel NUC," https://en.wikipedia.org/wiki/Next_Unit_of_Computing [Online; accessed 1-Feb-2018].
[51] P. N. Mahalle, B. Anggorojati, N. R. Prasad, R. Prasad *et al.*, "Identity authentication and capability based access control (iacac) for the internet of things," *Journal of Cyber Security and Mobility*, vol. 1, no. 4, pp. 309–348, 2013.
[52] F. Bakir, R. Wolski, C. Krintz, and G. S. Ramachandran, "Devices-as-services: Rethinking scalable service architectures for the internet of things," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotedge19/presentation/bakir
[53] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management – part 1: General (revision 4)," *NIST Special Publication Revision*, 01 2016. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final
[54] J. Jonsson and B. Kaliski, "Rfc3447: Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1," Network Working Group, The Internet Society, Tech. Rep., 2003.
[55] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *International Workshop on Public Key Cryptography*. Springer, 2006, pp. 207–228.
[56] C. F. Kerry and C. R. Director, "Fips pub 186-4 federal information processing standards publication digital signature standard (dss)," 2013. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/186/4/final
[57] B. Pearson, L. Luo, Y. Zhang, R. Dey, Z. Ling, M. Bassiouni, and X. Fu, "On misconception of hardware and cost in iot security and privacy," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, May 2019, pp. 1–7.
[58] L. K. Authors, "Bpf documentation," https://www.kernel.org/doc/html/latest/bpf/index.html, 2021.
[59] *WebAssembly Specification*, WebAssembly Community Group, 2020, version 1.
[60] uBPF Authors, "ubpf," https://github.com/iovisor/ubpf, 2021.
[61] W. Authors, "Wasm3," https://github.com/wasm3/wasm3, 2020.
[62] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, Mar 2004, pp. 75–88.

Appendix A
Security discussion of CAPLets key exchange

In this appendix, we formally prove the security of our key exchange algoritm against passive, replay and man in the middle attacks. We begin by presenting our threat model. In our scenario, we consider the 2 legitimate parties A, a client, B, a server. We also consider an attacker C, who has full control over the network between A and B, and can manipulate, replay, drop, delay and inject packets at will.

CAPLets only protects client-server communications. As such, clients do not accept connections from servers or other clients, and only A can initiate a connection with B. Any prior, indirect, communication from B to A for token dissemination happened over a secure channel, e.g. using a TLS based protocol. C is assumed to have access to all network packets, including plain-text handshake and cipher-text messages between A and B either over TLS or the CAPLets protocol.

For a successful attack in CAPLets, C must recover or choose the shared tag known both to A and B. Compared to TLS, our handshake is much simpler and has very few messages and operations. To prove our algorithm correct, we must show that under our threat model, an attacker may not influence the parties to choose an encryption key under the control of the attacker by modifying and/or replaying packets.

There are 3 handshake messages: **X.** Client hello from A to B, containing $body(T_E)$, containing a fresh N_A.

Y. Server hello from B to A, containing $MAC(tag(T_E), N_A)||N_B$, where N_B is a fresh nonce.

Z. Client done from A to B, containing $MAC(tag(T_E), N_B)$.

We formally define our axioms as:

1) B is in posession of root token R
2) B is a server
3) A is in posession of T
4) T is derived from R, $T \in \mathcal{S}_R$
5) C is in posession of τ
6) τ is derived from R, $\tau \in \mathcal{S}_R$
7) τ is not T, $\tau \neq T$
8) T is not derived from τ, $\tau \notin \mathcal{P}_T$
9) N_A and N_B are fresh, i.e. were never used before
10) The MAC function is strong

Lemma A.1. *C cannot recover $tag(T_E)$ given message **X**.*

Proof. Assume C can recover $tag(T_E)$ given τ and $body(T_E)$. This implies $T_E \in \mathcal{S}_\tau$. However, since T_E in $\mathcal{S}_T$, this implies $\tau = T \vee \tau \in \mathcal{P}_T$, contradicting axioms 7 and 8. ■

Lemma A.2. *C cannot recover $tag(T_E)$ given messages **X**, **Y** and **Z**.*

Proof. Assume there exists an inverse MAC function such that $MAC(MAC^{-1}(tag, body), body) = tag$), thus recovering the secret of the MAC function given a body and the corresponding tag. C can recover $tag(T_E) = MAC^{-1}(MAC(tag(T_E), N_x), N_x)$ where $x \in \{A, B\}$. However, the existence of MAC^{-1} contradicts axiom 10. ■

Remark. It follows from Lemmas A.1 and A.2, this algorithm is secure against passive attacks.

Lemma A.3. *C cannot reliably replay previous communications without detection by A or B.*

Proof. Assume C can replay packets that with high probability pass the challenge on either side. Implying there exists a communication with parameters $T_E{}', N_A{}', N_B{}'$ such that $MAC(tag(T_E), N_x) = MAC(tag(T_E{}'), N_x{}')$ where $x \in A, B$. Note that T_E contains N_A.

There are two possibilities:

1) $tag(T_E) \neq tag(T_E{}') \vee N_x \neq N_x{}'$: as $p(MAC(tag(T_E), N_x) = MAC(tag(T_E{}'), N_x{}')) \approx 1$ (probability of another communication having the same challenges as the current one is very high), this contradicts axiom 10 as collisions with a strong MAC function are rare.
2) $tag(T_E) = tag(T_E{}') \wedge N_x = N_x{}'$: implies reuse of nonces contradicting axiom 9.

■

Lemma A.4. *C cannot tamper with packets without detection by A or B.*

Proof. Assume C can produce packets such that $MAC(tag(T_E), N_x) = MAC(tag(U), M) \wedge N_x \neq M$ where $x \in A, B$. As such, C can change N_x to M to trick A or B into accepting a different nonce. This would be possible if $tag(T_E) = tag(U)$, however, since tags are computed with the MAC function, $T_E \neq U$ contradicts axiom 10. And $T_E = U$ contradicts lemma A.1. ■

Lemma A.5. *C cannot trick A and B to use different keys for the same communication by intercepting and replaying communications over 2 different channels*

Proof. Assume C can perform a valid handshake for channel A->C where A thinks it is A->B. Since A sets up the communication, it selects $tag(T_E)$ even before the communication, so C cannot convince A to use a different secret. To convince A that it is talking with B, C would have to produce $MAC(tag(T_E), N_A)$. If C was able to produce $tag(T_E)$, it would be a contradition to lemma A.1. Without producing $tag(T_E)$, C cannot decrypt packets later even if it can produce a valid message passing the challenge, which would actually contradict lemma A.4. ■

Remark. It follows from lemmas A.3, A.4 and A.5 that this algorithm is resistant to replay and man in the middle attacks.