Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution Over Untrusted Cache-enabled Networks

Moreno Ambrosin[1,⋆], Christoph Busold[2], Mauro Conti[1,⋆⋆], Ahmad-Reza Sadeghi[3], Matthias Schunter[4]

[1] University of Padua, Italy, {lastname}@math.unipd.it
[2] Intel CRI-SC, TU Darmstadt, Germany, christoph.busold@trust.cased.de
[3] CASED/TU Darmstadt, Germany, ahmad.sadeghi@trust.cased.de
[4] Intel Labs, schunter@acm.org

Abstract. Secure and fast distribution of software updates and patches is essential for improving functionality and security of computer systems. Today, each device downloads updates individually from a software provider distribution server. Unfortunately, this approach does not scale to large systems with billions of devices where the network bandwidth of the server and the local Internet gateway become bottlenecks. Cache-enabled Network (CN) services (either proprietary, as Akamai, or open Content-Distribution Networks) can reduce these bottlenecks. However, they do not offer security guarantees against potentially untrusted CN providers that try to threaten the confidentiality of the updates or the privacy of the users. In this paper, we propose Updaticator, the first protocol for software updates over Cache-enabled Networks that is scalable to billions of concurrent device updates while being secure against malicious networks. We evaluate our proposal considering Named-Data Networking, a novel instance of Cache-enabled overlay Networks. Our analysis and experimental evaluation show that Updaticator removes the bottlenecks of individual device-update distribution, by reducing the network load at the distribution server: from linear in the number of devices to a constant, even if billions of devices are requesting updates. Furthermore, when compared to the state-of-the-art individual device-update mechanisms, the download time with Updaticator is negligible, due to local caching.

Keywords: Software Updates, Secure Updates Distribution, Attribute-based Encryption, Internet of Things, Cache-enabled Network

⋆ Corresponding author.

⋆⋆ Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission under the agreement n. PCIG11-GA-2012-321980. This work has been partially supported by the TENACE PRIN Project 20103P34XC funded by the Italian MIUR. Part of this work has been performed while Mauro Conti was visiting TU Darmstadt thanks to a German DAAD fellowship.

1 Introduction

The growing diffusion of electronic devices creates new issues and challenges. Consider billions of lighting devices [29], embedded controllers, or mobile and wearable devices. More generally, the so-called Internet of Things is extending the Internet to billions of devices that need to be connected and updated. One of the resulting challenges is efficient and secure distribution of software updates to these devices. According to the 2013 US-CERT Security Alerts [1], most of the new software vulnerabilities can be resolved by applying software updates. Hence, fast and secure delivery of software updates plays a key role in securing software systems. In particular, once a vulnerability is published (e.g., see the case of the recent SSL "Heartbleed" vulnerability [18]), the system becomes exposed to a large base of potential adversaries. Here, a fast update is fundamental.

Most of the existing remote update protocols focus on ensuring integrity and authenticity of the transmitted updates, i.e., they guarantee that only untampered updates from a legitimate source will be installed on the device. However, in many cases software updates are required to be confidential. Examples include protection of embedded software against reverse-engineering or the distribution of valuable map updates in automotive systems and portable devices. A simplistic approach to achieve confidentiality for updates is securing the communication between client and software provider server applying end-to-end encryption (e.g., using SSL [31]). Each client device then requests and downloads the latest available update directly from the software provider, encrypted and signed by the software update source. Although this approach guarantees confidentiality and authenticity of software updates, it is not suitable for large-scale systems, since it would not scale due to the load on the software distribution servers, which is increasing linearly in the number of devices.

To mitigate this efficiency problem, software providers usually rely on third-party distribution infrastructures [5], e.g., Content Delivery Networks (CDNs) such as Akamai [2] or Windows Azure CDN [33]. These infrastructures apply in-network caching and replication strategies in order to speed-up content distribution. However, by using third-party distribution networks, software providers can no longer apply end-to-end encryption. Instead, they must allow point-to-point encryption between client devices and the distribution network, and between the distribution network and the software provider [3]. This poses security and privacy issues, since transferred updates are cached unencrypted by each distribution network server, and the software provider or the distribution nodes know which device (or user) is asking for what.

Our Contribution. In this paper, we propose a new solution for efficient distribution of confidential software updates that is scalable and optimized for untrusted distribution media which support in-network caching. The contributions of this paper are threefold:

i) We present Updaticator, a protocol for efficient distribution of confidential software updates, optimized for untrusted cache-enabled distribution media.

The protocol reduces bandwidth consumption and server load, provides end-to-end security, and is scalable to billions of devices. To enable caching, one main goal is to encrypt each given update under a corresponding symmetric key to ensure identical ciphertexts for all devices receiving this update. These keys are then distributed using Attribute-based Encryption (ABE).

ii) We define a system model and security requirements for this class of protocols and analyze the security of the proposed protocol.

iii) We describe a prototype implementation of our protocol on Named-Data Networking [21] as broadcast medium for efficient and secure distribution of updates, and evaluate its performance.

Organization. The remainder of the paper is organized as follows. In Sec. 2 we describe the system model and the security requirements for the design of Updaticator. In Sec. 3 we introduce the primitives used in the description of our protocol. In Sec. 4 we introduce the Updaticator protocol while in Sec. 5 we provide a security analysis, based on the security requirements introduced in Sec. 2. In Sec. 6 we present an experimental evaluation, which demonstrates the benefits of our solution. Finally, in Sec. 7 we analyze current state-of-the-art approaches to updates delivery and differentiate our results. Eventually, we conclude in Sec. 8 and describe possible future work.

2 System Model and Requirements

In this section we introduce the system model, on which we base our work, and derive the security requirements for our software updates distribution protocol.

2.1 System Model

In our model, groups of *clients* request software updates from a specified update source. As shown in Fig. 1, we assume the presence of an *Update Server* (*US*), as introduced in [14]. Each client queries *US* to retrieve information about the latest available update package. Furthermore, we assume the presence of a *Distribution Server* (*DS*), which is responsible for efficient dissemination of updates. For the sake of generality, we also assume the presence of a *Policy Server* (*PS*), that generates keys, imports updates into the system, and defines the update policy, i.e., which update should be provided to which device.

2.2 Security Requirements

We now define the desirable security requirements for Updaticator.

Confidentiality. Our solution for update distribution must be able to guarantee the confidentiality of updates, since we assume the distributed updates to be proprietary data. This means that each client should be able to decrypt a software update if and only if it has been authorized by *PS*.

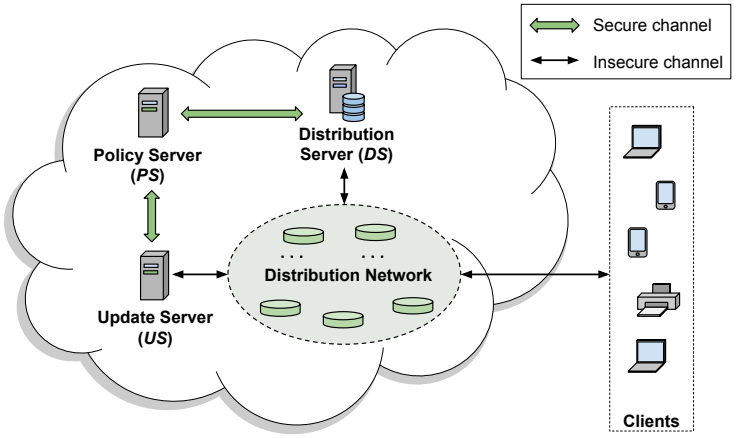


Fig. 1: System model.

Authenticity and Integrity. Our solution for update distribution must guarantee the possibility for all the clients to verify the integrity and authenticity of the downloaded software updates, in order to prevent attackers from replacing legitimate software updates with malicious code.

Freshness. Network caches reduce network traffic and server load. However, it is possible that device requests are satisfied by outdated updates still in cache. Furthermore, attackers could intentionally mask the presence of new updates in order to prevent devices from patching security issues. For this reason, our solution should provide clients with the means to verify whether the answer to its update request is fresh, i.e., corresponds to the most recent update released by PS. We achieve this by introducing a freshness interval Δt that defines the maximum age of the latest update information.

2.3 Adversary Model

Informally, we consider the following attack scenarios:

(1) Legitimate devices could try to obtain software updates which are not intended for them.
(2) Network attackers could try to get access to confidential updates or compromise devices by injecting unauthorized or modified updates.

More formally, the adversary model is defined as follows: the policy server PS is an internal server that feeds information to the externally-facing servers US and DS. We assume that the update infrastructure consisting of US and PS is secure (including the internal communication between US and PS) and trusted by all devices that are updated by these servers. For the confidentiality of a given update, we assume that the client devices receiving this update neither

reveal the update packages nor their private keys. As a consequence, we cannot guarantee confidentiality of updates that are targeted to compromised devices.

In order to allow our solution to scale to a huge number of devices, we do not consider revocation of individual devices or keys. Revocation is important for broadcast media, where cloned subscriber cards pose a high risk. For software updates, in contrast, rebroadcasts of decrypted updates are more likely. This risk cannot be mitigated by revocation alone, since it further requires traitor tracing in order to identify the key that should be revoked. Traitor tracing techniques such as watermarking, however, require individualized updates, which would prevent caching and thereby compromise the scalability of our scheme.

The update distribution is carried out over an untrusted network, therefore neither the update infrastructure nor individual devices trust DS. The considered attackers are Dolev-Yao [13] adversaries that have full control over the communication channel and can eavesdrop, manipulate, inject and replay messages between any device and the update infrastructure.

3 Background

We now provide some background knowledge, introducing Attribute-based Encryption, and Named-Data Networking.

3.1 Attribute-based Encryption

Attribute-based Encryption (ABE), first proposed by Sahai and Waters in 2005 [30], is a type of public-key encryption that allows fine-grained data access control based on *attributes*. With ABE, the data owner defines an *access policy* w, i.e., a combination of attributes that a legitimate user must own in order to access the data. An access policy can be represented as a Boolean expression, specifying the attributes required to access the data. For example, suppose we define three different attributes, *Student*, *MSc*, and *Professor*. If we want to make some data accessible only to users that are professors or MSc students, a possible access policy can be expressed as $\{\{Student \wedge MSc\} \vee Professor\}$.

In our work, we consider Ciphertext-Policy Attribute-based Encryption (CP-ABE), an Identity-based Encryption scheme first introduced by Bethencourt et al. [6], and then refined and extended in several other works [36], [35]. With CP-ABE, the access policy is bound to the ciphertext, while users' private keys are generated based on the users' attributes. CP-ABE allows the definition of high-level policies, and therefore is particularly useful in scenarios where an entity wants to restrict the access of a piece of information only to a subset of users within the same broadcast domain [15]. Moreover, CP-ABE is resistant against collusion attacks by design [6].

A generic CP-ABE scheme provides the following four basic algorithms:

- SETUP(). This algorithm generates public key pk^{ABE} and master secret mk^{ABE}.

- KeyGen($mk^{ABE}, Attr^C$). This algorithm takes as input the master key mk^{ABE} and the user attribute list $Attr^C$, and outputs the user's private key $sk^{ABE,C}$.
- ABEnc$_{pk^{ABE},w}(m)$. The encryption algorithm takes the public key pk^{ABE}, the specified access policy w, and the message m as input. It outputs a ciphertext that can only be decrypted by a user with an attribute list $Attr^C$ such that $Attr^C$ satisfies the access policy w.
- ABDec$_{sk^{ABE,C}}(c)$. The decryption algorithm takes as input the public key pk^{ABE}, the private key $sk^{ABE,C}$ of user C, and the ciphertext c. It returns the plaintext m if and only if user attribute list $Attr^C$ satisfy the access policy w.

3.2 Named-Data Networking

Named-Data Networking (NDN) [21] is a new Internet architecture optimized for efficient content distribution. NDN is an instantiation of the Content-Centric Networking (CCN) approach [19], in which data is accessed by name instead of location, and the routing is based on content names. In NDN, each content is bound to a unique hierarchically-structured name, formed by different components separated by "/". As an example, a possible name for the opinions webpage of the CNN website is `/cnn/politics/opinion`, while `/cnn/politics/` is a *name prefix* for that name.

Communication in NDN is *consumer-driven*: each consumer requests data by issuing *interest packets*, which are then satisfied by *data packets* provided by content *producers*. When a consumer sends a request for a particular content, the corresponding interest is forwarded to the first-hop router. Each NDN router maintains two lookup tables: Pending Interest Table (PIT) and Forwarding Information Base (FIB). PIT is used to keep track of all the pending requests for a given content name. Each entry of the PIT is in the form $< interest, arrival\_interfaces >$, where $arrival\_interface$ is the set of the router's interfaces to which the interest have been already forwarded. FIB is populated by a name-based routing protocol, and used by routers to forward outgoing interests to the right interface(s). When a router receives an interest, it first checks its PIT to determine whether another interest for the same name is currently outstanding. If the same name is already inside the PIT, then the interest arrival interface is searched inside the corresponding $arrival\_interfaces$ set. If the router finds a match, the interest is discarded, otherwise, the new interface is added to $arrival\_interfaces$ set, but the interest is not forwarded. If no matching entry was found in the PIT, a new PIT entry is created, and the interest is forwarded based on the FIB table. Once received the interest, the producer of the content injects a matching data packet into the network, thus *satisfying* the interest. The data packet is then forwarded towards the consumer, traversing, in reverse, the path of the corresponding interest.

An important feature of NDN is distributed caching, which is intended to reduce traffic and load of the network. Once a data packet is received by a router, it is stored inside its local cache, named Content Store (CS), according

to some cache policies. In this way, all subsequent interests matching the same data packet previously stored inside the CS, will be immediately satisfied with the local copy of the content, thus being no longer forwarded.

Most currently existing implementations of NDN are built as an overlay on top of the TCP/UDP transport protocols, e.g., NDNx [23]. This allows easy integration with the current Internet infrastructure.

4 Updaticator: Our Scalable Update Protocol with End-to-End Security

We now describe Updaticator, our solution for scalable and secure software updates distribution over Cache-enabled networks. Our protocol comprises three different phases:

(1) *Update publication.* In this phase, a new available update is published. The Policy Server (PS) generates and sends the access policy for the update package, and a new random encryption key for this update, to the Update Server (US). Then, PS sends the encrypted package to the Distribution Server (DS), which takes care of its distribution.
(2) *Update selection.* In this phase, a client (C) checks for the presence of new updates, issuing a request to US. The information is used to eventually retrieve a new software update.
(3) *Update retrieval.* In this phase, C downloads the update package, issuing a request to DS.

Without loss of generality, we now provide a detailed description of each phase of our protocol on top of NDN.

Notation. In the remainder of this paper, we assume that both PS and US have a key pair, (sk^{PS}, pk^{PS}) and (sk^{US}, pk^{US}), respectively, that they can use for generating signatures. We refer to pkg_u as the software update package, and to id_u as its identifier, calculated as $\textsc{Hash}(pkg_u)$, where Hash is a collision-resistant hash function. Moreover, we suppose that each software update package pkg_u has an associated access policy w_u, set by PS. Each client C has an attribute list $Attr^C$, which is represented by his private key $sk^{ABE,C}$. We indicate with k_u the symmetric key used to encrypt pkg_u, and with $\textsc{Enc}_{k_u}$ and $\textsc{Dec}_{k_u}$ symmetric encryption and decryption functions, respectively. We also indicate with $\textsc{Sign}_{sk^X}$ the computation of a signature using secret key sk^X, and with $\textsc{Verify}_{pk^X}$ the verification of a signature.

On Scalability. In typical deployments, devices are connected to the Internet via a gateway with limited bandwidth. Our main goal is to ensure scalability to billions of devices, i.e., to ensure that the network load on the distribution server as well as the network load on the gateway is constant in the number of devices and only depends on the stable number of available updates. This is achieved by ensuring that all the phases of our protocol, involving client devices, are

non-interactive and cacheable. Regarding the complexity of CP-ABE policies, each device class can be targeted with one instance of the CP-ABE scheme. Furthermore, each device usually has a limited set of (licensing) options that determine the set of updates (and corresponding keys) this device may receive. Therefore the complexity does not increase with the number of devices.

4.1 Update Publication Phase

The software update publication phase of Updaticator is presented in Fig. 2. When a new software update package, pkg_u, is released, PS computes its identifier id_u as the hash value of pkg_u, and generates a new key k_u, for symmetric encryption.

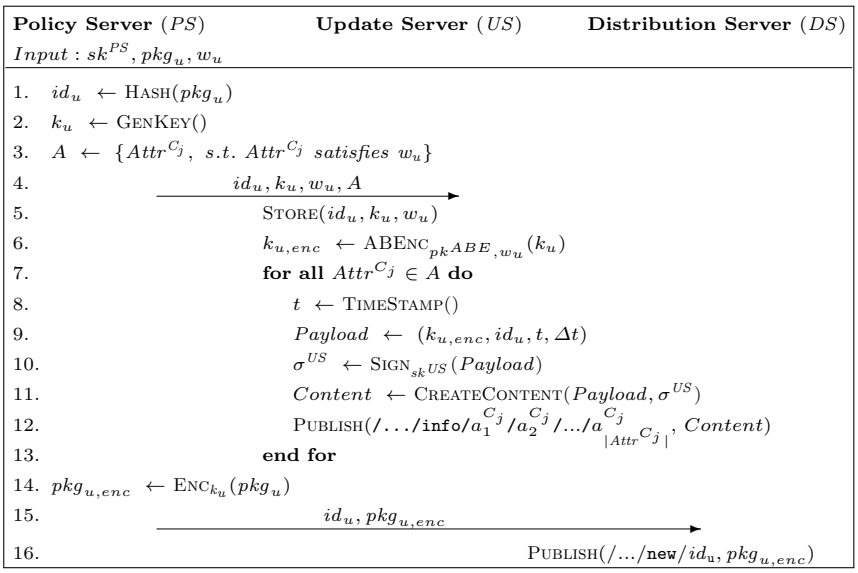


Fig. 2: Update Publication.

Then, PS associates an access policy w_u to pkg_u, creates the set A of all the existing attribute combinations that match with w_u, and finally forwards the tuple (id_u, k_u, w_u, A) to US, where it is stored inside a database (Fig. 2, lines 1-5). After that, US proceeds with the publication of the symmetric key k_u. Here, the key idea is to allow scalable distribution of the encryption key k_u, at the same time taking advantage from in-network caching provided by the NDN distribution network, and providing fine-grained access control to k_u. We achieve this with the aid of CP-ABE. In our solution, each client has an attribute-specific decryption key $sk^{ABE,C}$ corresponding to its set of attributes $Attr^C$, while US has the public key pk^{ABE} that is used for encryption. Then, for each new software update package pkg_u, US encrypts the key k_u with the public key pk^{ABE}, together

with the access policy w_u (Fig. 2, line 6). *US* records the current timestamp t and determines the time interval Δt within which the update information should be considered *fresh* by clients. This will allow clients to verify the *freshness* of the retrieved information (Fig. 2, lines 8-9). Finally, *US* produces a signature σ^{US} of id_u, k_u, t and Δt (Fig. 2, line 10), and publishes the NDN content $data = (id_u, k_{u,enc}, t, \Delta t, \sigma^{US})$, according to a specific naming scheme (Fig. 2, line 12). A possible example is the following. Suppose $Attr^C = [a_1^C, a_2^C, ..., a_n^C]$ being the list of attributes of client C. *US* distributes *data* under the NDN name `/.../info/`a_1^C`/`a_2^C`/.../`a_n^C, for each $Attr^C$ that satisfies the access policy w_u. Let A be the set of all m different possible combinations of client attributes that satisfies the access policy w_u. *US* will publish the content *data* under m different names, thus treated by NDN as m different contents. Let $size(data)$ be the size of the packet *data*. In the worst case, the same content will be cached m times by the distribution network, with a theoretical maximum cost for the entire caching network, in term of space, of $m \cdot size(data)$. However, optimized caching policies could reduce the space needed to cache contents. For example, a possible improvement is the adoption of the following caching policy: each data packet is decomposed into payload p_{data}, header h_{data} and packet signature sig_{data}. Then, the data packet payload p_{data} is cached if and only if p_{data} is not already present in router cache, while header and signature are always stored. This caching policy could reduce the required caching space for contents with the same payload that are distributed under different names, to a maximum of $m \cdot [size(h_{data}) + size(sig_{data})] + size(p_{data})$. The update package pkg_u is finally encrypted by *PS* with the symmetric key k_u (Fig. 2, line 14), and published by *DS* under the NDN name `/.../new/`id_u (Fig. 2, line 16). In this way, each interest issued by the client, for the software update package identified by id_u, will be satisfied either by *DS* directly or by a NDN router within the distribution network, with a cached copy of $pkg_{u,enc}$. Note that the communication between *PS*, *US* and *DS* does not impose a specific communication protocol to be used. We only assume that this communication is secure (e.g., via the SSL/TLS protocol).

4.2 Update Selection Phase

The update selection phase of Updaticator is presented in Fig. 3. In order to obtain information on the latest available update, a client C sends a request specifying its attributes $Attr^C$, i.e., issuing an interest for `/.../info/`a_1^C`/`a_2^C`/.../`a_n^C (Fig. 3, line 1). The interest is forwarded to the NDN distribution network, which either satisfies it with a matching copy of the required content, or forwards the interest up to *US*, if no matching contents are available. In the latter case, the information is then stored by the routers on its path back to the client (Fig. 3, lines 2-5).

Upon receiving the response, the client first verifies the freshness of the received content, i.e., if the software update information is outdated (Fig. 3, line 6). This is done checking if the value $t + \Delta t$ is greater than the current time. If the information is fresh, C proceeds by verifying the signature σ^{US} and checking if id_u is already contained in $UpdatesList$, the list of all the updates previously

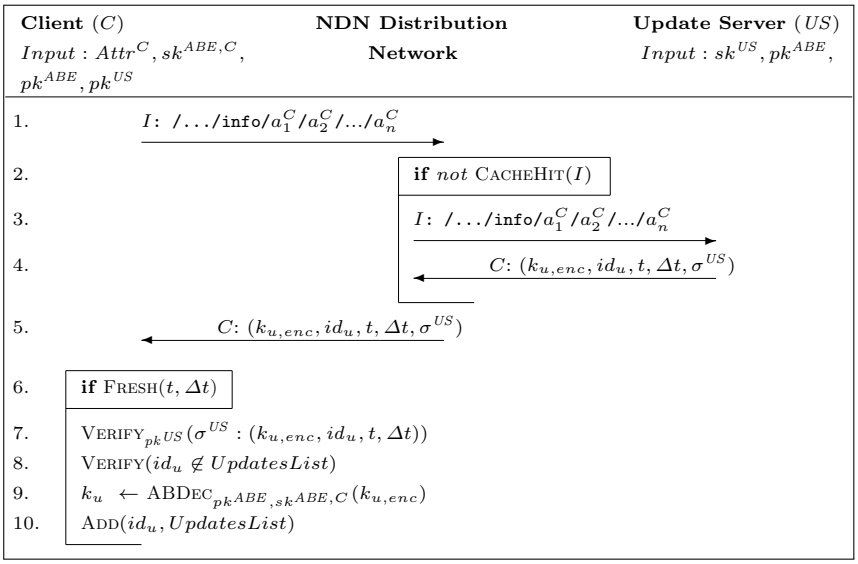


Fig. 3: Update Selection.

installed by C (Fig. 3, lines 7-8). If all these tests pass, C decrypts $k_{u,enc}$ with its secret key $sk^{ABE,C}$ and finally adds id_u to $UpdatesList$ (Fig. 3, lines 9-10).

In case either the authentication or the freshness verification fails, C will repeat the update selection procedure, this time requesting the update information directly from US. In NDN, this can be achieved by setting the `AnswerOriginKind` parameter of the interest packet to 0 [22]. In this way, NDN routers will never satisfy the interest with cached content, but route it up to US instead. It should be noted that in this case, the newly received message is expected to be fresh, since it should originate directly from US. Otherwise the client will conclude that it is under a DoS attack. Similarly, if the response is not authentic, the client can detect the presence of a possible DoS attack, which prevents the client from downloading new updates. Finally, only in the case in which the newly received response is authentic and fresh, and $id_u \notin UpdatesList$, the client will conclude that there are no available updates. Since any client whose attributes match the access policy w_u, can decrypt the response $data$, the same content can be cached by the network and served to all the clients with the same attributes.

4.3 Update Retrieval Phase

Fig. 4 shows the Updaticator update retrieval phase. After obtaining a valid update identifier id_u and the corresponding symmetric key k_u, the client can download the encrypted update package $pkg_{u,enc}$ from DS, specifying only the update identifier id_u, i.e., issuing an interest for `/.../new/`id_u (Fig. 4, line 1). Similar to the update selection phase, the interest is forwarded to the NDN

distribution network, which either satisfies it with cached content matching the interest, or forwards it up to *DS* (Fig. 4, lines 2-5). After receiving the encrypted software update package, the client can decrypt it and verify its integrity by comparing $\textsc{Hash}(pkg_u)$ with id_u (Fig. 4, lines 6-7).

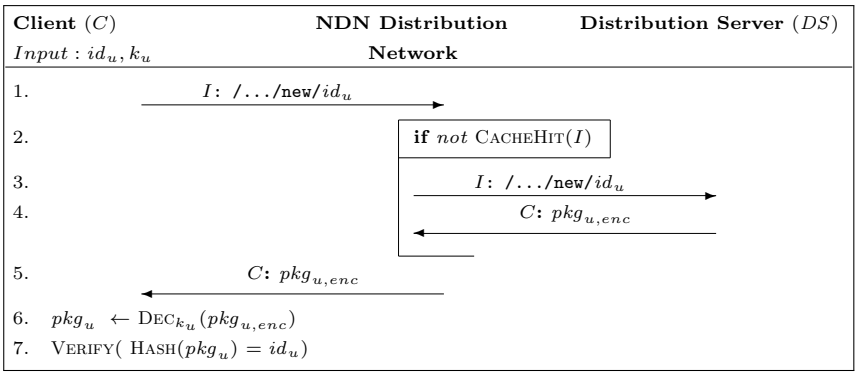


Fig. 4: Update Retrieval.

It should be noted that the probability of the request hitting a cache closer to the client increases with the number of clients downloading the same software update. Finally, for low-memory devices the encryption of the distributed software updates can be performed with the encryption technique proposed by Nilsson et al. in [25]. This technique splits the update in fragments. In reverse order, for each fragment a hash is computed and the hash of each fragment is stored together with the following fragment, i.e., forming a hash chain. Only the fragment containing the information about the update, together with the hash of the first fragment, is signed. In this way, a client can verify the authenticity and integrity of the first fragment, and of all the other fragments verifying the hash chain, thus allowing "load-and-install" of the update.

5 Security Analysis

We now provide a security analysis regarding the requirements from Sec. 2.2.

Confidentiality. Updaticator provides software update confidentiality through the use of symmetric encryption. During the update publication phase of our protocol, *PS* generates a key k_u and associates it with the new software update package pkg_u. Since each key k_u is randomly chosen, we can assume that it is unique for each pkg_u. Following the assumption that the publication infrastructure is secure, during this phase an attacker can neither extract k_u from *PS* or *US*, nor access the unencrypted update package. Furthermore, only the encrypted package $pkg_{u,enc}$ and the hash of pkg_u, i.e., id_u, are transmitted or

published during each phase. As a consequence, since the cryptographic primitives are assumed to be secure, both values do not allow the attacker to gain information on the unencrypted update package or key k_u. The confidentiality of the update package can therefore be reduced to the confidentiality of k_u, i.e., an attacker can obtain the update package if and only if she is in possession of k_u, which she only can obtain through the update selection protocol. The update selection phase uses CP-ABE in order to distribute k_u. Each client has a decryption key sk^{ABE}, which matches exactly its set of attributes. Only clients with a matching list of attributes will be able to decrypt it. We can conclude that the confidentiality of each software update is assured.

Authenticity and Integrity. During update publication phase, the update identifier id_u is computed as the hash value of pkg_u. Then, during the update selection phase id_u is provided by *US* in response to each device request inside a signed message. Hence, the client can verify the authenticity of id_u. As the client verifies the hash of the package against id_u during the update retrieval phase of Updaticator, and the hash function is collision-resistant, the authenticity and integrity of the update package depends solely on the authenticity and integrity of id_u. Since *US* is trusted and the signature scheme is assumed to be secure, id_u is authentic and the client can conclude that the package is authentic as well.

Freshness of the Interactive Update Selection Protocol. During update selection phase, the information about the latest software update available is distributed in a cacheable form, i.e., it is intended for offline updates and is identical for a class of clients. Consequently, such a response can also be provided by caches or by offline media. When *US* first publishes a new available update package pkg_u, it also specifies the publication timestamp t and a time interval Δt, that indicates the time interval in which a client should consider the information fresh. A client can verify information freshness by checking its current timestamp against $t+\Delta t$. If the content is not fresh, i.e., not received before $t+\Delta t$, the client, once verified data authenticity, requests the same information directly from *US*. For this reason, we conclude that Updaticator guarantees content freshness.

6 Prototype and Evaluation

We now document our prototype and evaluation of the proposed update scheme. While arbitrary Cache-enabled Networks can be used, we benchmarked our scheme using a distribution network built with Named-Data Networking as an overlay network on top of TCP/IP. In this scenario, NDN nodes can be distributed over the Internet and can be used for cached distribution of arbitrary content including updates.

6.1 Evaluation Setup

In order to provide a large-scale evaluation of the use of Named-Data Networking (NDN) to build a network for efficient software update distribution, we carried

out our tests on the ns-3 simulator [24]. We focused on update requests generated by always-on-line devices, which check for updates periodically. We assumed that update requests are generated in bursts, i.e., n devices check for updates and/or download an update within a *time window interval* $tw = [t_u, t_u + \Delta t]$.

We compared content download via the HTTP protocol and a distribution network built with NDN as a TCP/IP overlay. In the former case, we published the content through an `thttpd` Web Server [32], while in the latter case, we published the content on an NDN repository built using the NDNx protocol [23]. The integration of such an application with the ns-3 simulator has been achieved by leveraging the DCE module [12] of ns-3. Our experiments were carried out on a DFN-like topology [11], depicted in Fig. 5.

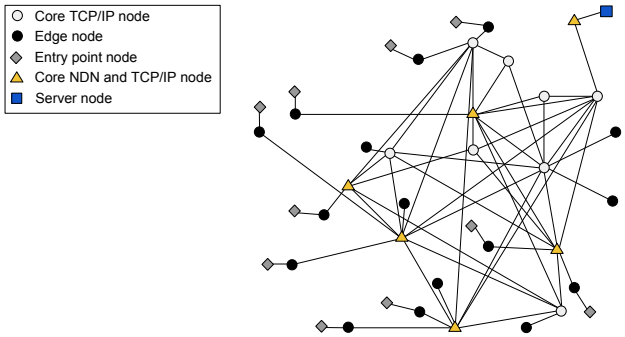


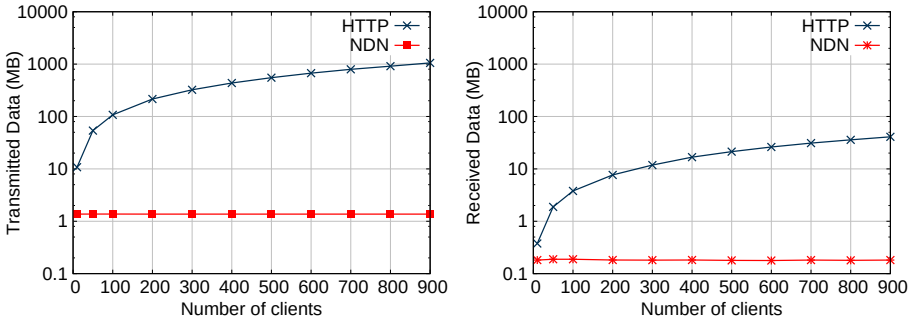
Fig. 5: DFN-like network topology.

In our simulated topology, we introduced three different types of nodes. *Core nodes* represent the main part of the topology and are connected through 1 Gbps point-to-point links. We introduced two types of core nodes: NDN-capable core nodes (triangles in Fig. 5), which can communicate via the NDN protocol, and simple TCP/IP core nodes (white circles in Fig. 5). *Edge nodes* (dark circles in Fig. 5) are used to access the network. In our simulation, edge nodes are connected to core nodes through 100 Mpbs links. *Entry nodes* (rhombus in Fig. 5) are nodes to which all clients are connected. Each client can perform both HTTP and NDN requests.

We considered an increasing number of clients, from 100 to 900, connected on the entry point nodes. Each device requested a content of 1 MB, starting the download at a uniformly chosen time t in $tw = [t_u, t_u + 30]$ seconds. We then measured the average Content Retrieval Time (CRT) for the devices and the total amount of traffic at the server side. Moreover, in order to provide a complete vision of the advantages in adopting a Cache-enabled Network, we analyzed the bandwidth utilization.

6.2 Network Load on the Update Server

The first result of our analysis was that our approach indeed reduces the network load on the update server from linear (or worse) to just a constant level (Fig. 6). This is achieved by the caching of updates over the NDN distribution network so that the server only needs to deliver a small number of original copies of each update to populate the caches. Fig. 6a and Fig. 6b depict average data traffic sent and received by the server. Results are shown for a time window of 30 seconds. The results show that the use of a Cache-enabled Network as NDN highly reduces the load at the server side from linear (please note that graphs in Fig. 6a and Fig. 6b are represented in log-scale) to constant, thus allowing system scalability and preventing DDoS attacks against *DS*.



(a) Sent data. (b) Received data.

Fig. 6: Sent and received data by the server, NDN vs. direct download (content size 1 MB; time window of 30 s; log-scale transforms linear growth into a log curve).

6.3 Time Required to Retrieve an Update

The introduction of a Cache-enabled Network also reduces the average Content Retrieval Time (CRT) (Fig. 7) to a constant even for large numbers of devices. The traditional scheme without caching only performs for a small number of devices. For larger numbers the performance drops dramatically. Furthermore, under load the individual times vary within in a wide range. This makes the individual update times largely unpredictable.

Crypto Performance. For performance testing purposes, we adopted the `cpabe` library [9], which provides an implementation of the CP-ABE scheme proposed by Walters et al. [6], and the `openssl` library [27]. We selected AES-CBC, with key size of 256 bits for the symmetric encryption of the software update package and the symmetric key k_u, and adopted the RSA algorithm with a 4096 bits key

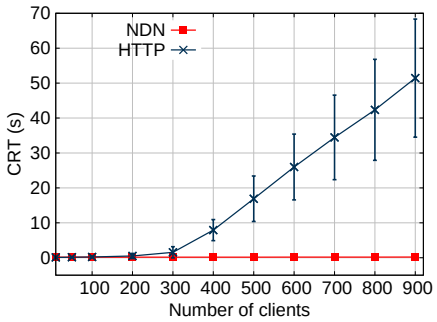


Fig. 7: Average time needed to retrieve a content of size 1 MB via NDN vs. direct download via HTTP for 900 clients and 1 MB content download; time window $t_{MAX} = 30$.

for content signing. Moreover, we considered SHA256 and SHA384 for hashing. Our tests have been conducted on a system equipped with two 2.4 GHz Intel Core^{TM} 2 Duo CPUs and 4 GB RAM and a 256 bit key k_u for AES-CBC. Results are reported in Table 1.

Function	**Time**	**Function**	**Time**
CP-ABE encrypt	77.47 ms	CP-ABE decrypt	32.62 ms
AES-256-CBC encrypt	28.77 ms	AES-256-CBC decrypt	26.61 ms
RSA signature create	31.64 ms	RSA signature verify	5.20 ms
SHA256	13.3 ms	SHA384	10.99 ms

Table 1: Evaluation of the cryptographic primitives used in our simulation (AES, SHA256 and SHA384 with 1 MB input; CP-ABE with 65 bytes input and 5 attributes; signatures with 438 bytes input; average numbers over 10,000 runs.

6.4 Power Consumption of the Client Devices

Reducing the network load and required computation is essential in order to limit power consumption. This is particularly true for resource-constrained devices. The introduction of the freshness interval "Δt" allows each device to remain off-line most of the time (only checking for updates once in each time interval), and also provides the possibility for software distributors to determine the best tradeoff between client device power consumption and software update freshness.

7 Related Work

In this section we provide an overview of previous work related to secure software update distribution. Bellissimo et al. [5] provide a security analysis of existing update systems, which reveals several weaknesses such as vulnerabilities against man-in-the-middle attacks. This emphasizes the importance of secure software update distribution mechanisms.

We focus on encrypted updates that can be cached. Due to space limitations, we do not survey non-cacheable updates distribution protocols such as the one proposed in [14], or mechanisms that do not provide encryption such as the one in [8]. Instead, we contrast our work to related work on updates using broadcast encryption as the underlying key management mechanism. Adelsbach et al. [4] propose to use broadcast encryption to distribute confidential software updates to embedded controllers inside an automotive system. Therefore their solution provides confidentiality and at the same time enables caching of updates, since the encrypted update is identical for all devices. Misra et al. [20] propose to use broadcast encryption to ensure confidentiality of a Content-Distribution Network built on top of NDN. Unfortunately, in this work the authors do not provide a cacheable "content selection" mechanism.

Similarly to OMA DRM [26], our scheme allows cacheable distribution of content (by separating the encrypted object from the associated decryption key). However, in order to retrieve the decryption key and the associated access policy, OMA DRM requires each client to establish an interactive session with the Policy Server. By using CP-ABE, our protocol does not require this interaction hence allowing a cacheable and scalable distribution of the decryption key.

We believe that broadcast encryption schemes (such as the ones in [7] and [17]) are not suitable for our system model. Indeed, adopting a broadcast encryption scheme, the Policy Server would need to define a cryptographic broadcast system for each group of devices, hence complicating the key management for both the Policy Server and the devices. Moreover, public-key broadcast encryption schemes do not provide constant size encryption keys, and therefore do not scale to billions of devices. Finally, while the broadcast encryption scheme proposed in [10] achieves constant size keys, the group management and the encryption must be carried out by the same entity. This is a limitation in our system model, since the creation and management of the device group (usually done by the device vendor) and the encryption of a targeted update (usually done by software vendor) could no longer be separated. In contrast, CP-ABE provides constant size keys, while allowing encryption using public parameters [16].

8 Conclusions and Future Work

Fast and secure software update distribution is a key issue in modern IT systems, particularly when the updates concern the fix of security vulnerabilities or are essential for business. As shown by our analysis, our approach is the first solution that makes large-scale updates practical for billions of devices. Future

work in this area will focus on the optimization of software update distribution for local networks and resource-constrained devices. Specific constraints coming from these environments call for novel solutions for these devices. In particular, those solutions should be able to coordinate different devices to maintain the compatibility among them (e.g., specifying constraints via policies), and being resilient to malicious devices that might hinder the success of the proposal.

Finally, we plan to provide revocation of individual devices. This aspect becomes an essential requirement in a stronger adversary model, where we consider compromised clients as possible attackers. Ostrovsky et al. [28] propose a CP-ABE scheme to specify revoked users directly inside the access policy of the ciphertext. However, the size of their policy grows linearly with the number of revoked clients. Therefore we are also looking into other possible solutions, e.g., hybrid schemes such as [35] and [34]. Furthermore, this requires a way to identify the source of a leaked update (traitor tracing), which is particularly challenging in our system model, since existing solutions prevent the cacheability of updates.

References

1. 2013 US-CERT Techical Security Alerts, `http://www.us-cert.gov/ncas/alerts/2013`
2. Akamai Content Delivery Network, `http://www.akamai.com`
3. Akamai Secure Content Delivery, `http://www.akamai.com/dl/feature_sheets/fs_edgesuite_securecontentdelivery.pdf`
4. Adelsbach, A. and Huber, U. and Sadeghi, A.-R.: Secure Software Delivery and Installation in Embedded Systems. In: Deng, R.H., Bao, F., Pang, H., Zhou, J. (eds.) ISPEC 2005. LNCS, vol. 3439, pp. 255–267. Springer, Heidelberg, (2005)
5. Bellissimo, A., Burgess, J., Fu, K.: Secure Software Updates: Disappointments and New Challenges. In: 1st USENIX Workshop on Hot Topics in Security, pp. 37–43. USENIX Association, Berkeley (2006)
6. Bethencourt, J., Sahai, A., Waters, B.: Ciphertext-Policy Attribute-Based Encryption. In: 2007 IEEE Symposium on Security and Privacy, pp. 321–334. IEEE Computer Society, Washington (2007)
7. Boneh, D., Gentry, C., Waters, B.: Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys. In: Shoup, V. (ed.) Crypto'05. LNCS, vol. 3621, pp. 258–275. Springer, Berlin, Heidelberg, (2005)
8. Cameron, D., Liu J.: apt-p2p: A Peer-to-Peer Distribution System for Software Package Releases and Updates. In: 28th IEEE Conference on Computer Communications, pp. 864-872. IEEE, New York (2009)
9. cpabe toolkit, `http://hms.isi.jhu.edu/acsc/cpabe/#documentation`
10. Delerablée, C., Paillier, P., Pointcheval, D.: Fully Collusion Secure Dynamic Broadcast Encryption with Constant-Size Ciphertexts or Decryption Keys. In: Takagi, T., Okamoto, T., Okamoto, E., Okamoto, T. (eds.) Pairing 2007. LCNS, vol. 4575, pp. 39–59. Springer, Berlin, Heidelberg, (2007)
11. Deutsches forschungsnetz (DFN), `https://www.dfn.de/en/`
12. Direct Code Execution (DCE), `https://www.nsnam.org/overview/projects/direct-code-execution/`
13. Dolev, D., Yao, A. C.: On the Security of Public Key Protocols. In: IEEE Transactions on Information Theory, vol. 29, issue 2, pp. 198–208. IEEE, New York, (1983)

14. Gkantsidis, C., Karagiannis, T., VojnoviC, M.: Planet Scale Software Updates. In: 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 423–434. ACM, New York, (2006)
15. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In: 13th ACM Conference on Computer and Communications Security, pp. 89–98. ACM, New York, (2006)
16. Guo, F., Mu, Y., Susilo, W., Wong, D.S., Varadharajan, V.: CP-ABE With Constant-Size Keys for Lightweight Devices. In: IEEE Transactions on Information Forensics and Security, vol. 9, issue 5, pp. 763-771. IEEE, New York, (2014)
17. Halevy, D., Shamir, A.: The LSD Broadcast Encryption Scheme. In: Yung, M. (ed.) Crypto'02. LCNS, vol. 2442, pp. 47–60. Springer, Berlin, Heidelberg, (2002)
18. Heartbleed SSL protocol vulnerability, `https://www.schneier.com/blog/archives/2014/04/heartbleed.html`
19. Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., Braynard, R. L.: Networking Named Content. In: 5th International Conference on Emerging Networking Experiments and Technologies, pp. 1–12. ACM, New York, (2009)
20. Misra, S., Tourani, R., Majd, N. E.: Secure Content Delivery in Information-centric Networks: Design, Implementation, and Analyses. In: 3rd ACM SIGCOMM Workshop on Information-centric Networking, pp. 73–78. ACM, New York, (2013)
21. Named-Data Networking Project (NDN), `http://named-data.org`
22. NDNx Documentation - Interest Message, `http://named-data.net/doc/0.1/technical/InterestMessage.html`,
23. NDNx – NDN protocol implementation, `http://named-data.net/codebase/platform/moving-to-ndnx/`
24. NS-3 Simulator, `https://www.nsnam.org/`
25. Nilsson, D. K., Roosta, T., Lindqvist, U., Valdes, A.: Key Management and Secure Software Updates in Wireless Process Control Environments. In: 1st ACM Conference on Wireless Network Security, pp. 100–108. ACM, New York, (2008)
26. Open Mobile Alliance. DRM Specification ver. 2.2, Technical Report (2011).
27. OpneSSL project. `https://www.openssl.org/`
28. Ostrovsky, R., Sahai, A., Waters, B.: Attribute-based Encryption with Non-monotonic Access Structures. In: 14th ACM Conference on Computer and Communications Security, pp. 195–203. ACM, New York, (2007)
29. Philips Hue, `http://meethue.com/`
30. Sahai, A., Waters, B.: Fuzzy Identity-based Encryption. In: Cramer, R. (ed.) Eurocrypt'05. LCNS, pp. 457–473. Springer, Berlin, Heidelberg, (2005)
31. Samuel, J., Mathewson, N., Cappos, J. and Dingledine, R.: Survivable Key Compromise in Software Update Systems. In: 17th ACM Conference on Computer and Communications Security, pp. 61–72. ACM, New York, (2010)
32. thttpd web server, `http://www.acme.com/software/thttpd`
33. Windows Azure, `http://www.windowsazure.com/en-us/`
34. Yu, S., Wang, C., Ren, K., Lou, W.: Attribute Based Data Sharing with Attribute Revocation. In: 5th ACM Symposium on Information, Computer and Communications Security, pp. 261–270. ACM, New York, (2010)
35. Zhiqian, X., Martin, K.M.: Dynamic User Revocation and Key Refreshing for Attribute-Based Encryption in Cloud Storage. In: 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 844–849. IEEE, New York, (2012)
36. Zhou, Z., Huang, D., Wang, Z.: Efficient Privacy-Preserving Ciphertext-Policy Attribute Based Encryption and Broadcast Encryption. In: IEEE Transactions on Computers, vol. PP, issue 99. IEEE, New York, (2013)