University of Modena e Reggio Emilia

XXXV cycle of the International Doctorate School in
Information and Communication Technologies

Doctor of Philosophy dissertation in
Computer Engineering and Science

Redesigning IT Systems to Guarantee Survivability

Federico Magnanini

Supervisor: Prof. Michele Colajanni
Co-supervisor: Luca Ferretti Ph.D.
Ph.D. Course Coordinator: Prof. Sonia Bergamaschi

Contents

List of Tables

List of Figures

Abstract - English

Modern society is increasingly dependent on reliable Information Technology (IT) services. This pervasive digitalization offers multiple benefits, but it allows attackers to tamper with systems and cause great damage to services, customers and citizens. Modern IT systems should be designed and implemented to achieve a minimum security level even after successful intrusions, but the state-of-the-art is not oriented to meet similar goals. In this thesis, we consider critical systems that cannot tolerate intrusions and propose original solutions to satisfy their *survivability*, that is, to guarantee security properties even in presence of successful attacks, failures, or accidents. The proposed ideas contribute to the fields of intrusion-tolerance and survivability in the context of access control, single sign-on authentication and software supply chains.

Abstract - Italiano

La società moderna dipende sempre più da servizi di Information Technology (IT) affidabili. Questa digitalizzazione pervasiva offre molteplici vantaggi, ma consente agli aggressori di manomettere i sistemi e causare gravi danni ai servizi, ai clienti e ai cittadini. I sistemi IT moderni dovrebbero essere progettati e implementati per raggiungere un livello minimo di sicurezza anche dopo intrusioni riuscite, ma lo stato dell'arte non è orientato ad offrire garanzie simili. In questa tesi, consideriamo sistemi critici che non tollerano intrusioni e proponiamo soluzioni originali per soddisfare la loro *survivability*, ossia per garantire proprietà di sicurezza anche in presenza di attacchi riusciti, guasti o incidenti. Le idee proposte contribuiscono ai campi della intrusion-tolerance e della survivability nel contesto del controllo degli accessi, dell'autenticazione single sign-on e delle catene di fornitura del software.

Chapter 1

Introduction

Information technology (IT) has changed the world to the extent that nowadays modern society depends on IT systems for several aspects of its operation. Indeed several sectors, from health care, to payments systems, telecommunications, or critical infrastructure, rely on IT systems to operate correctly and efficiently. However, while the widespread adoption of information technology in different domains certainly entails several advantages, its misuse allows to harm society in increasingly damaging ways. Such damage is demonstrated by recent serious incidents, which show that adversaries are becoming so effective at compromising IT systems that even if IT systems are designed to be secure against external threats, adversaries are able to compromise them nonetheless. Successful attacks against business IT infrastructure [39], software supply chains [92], or authentication systems [39, 4] are among the most notable examples of such remarkable adversary capabilities. Despite the individual differences among these attacks in terms of context and execution, their effectiveness lies in a common pattern shared by all mentioned incidents: in each attack the adversary was able to compromise an IT system component that was assumed to be trusted. As the whole IT system by definition relies upon trusted components to function correctly, trusted components are single points of failure, and thus their compromise inevitably has fatal consequences for the overall IT system security. As a result, the common pattern shared by these attacks highlights an urgent security challenge: to counter highly sophisticated adversaries it is no longer sufficient to design secure IT systems exclusively comprised of trusted components. Instead, IT systems should be designed to minimize the amount of trusted components, so that ideally a system consists mostly of untrusted components that cooperate to implement a given functionality, and only of very few trusted components that the whole system can assume to never be compromised. The benefits of this design is twofold. First, the adversary has less opportunities to exploit possibly fatal attacks against trusted components. Second, even if the adversary compromises a subset of untrusted components, the adversary only gains marginal advantage in subverting the whole system security. Both benefits increase attack costs and buy precious time for defenders to detect and thwart the attack. These guidelines allow to design IT systems that are able to survive successful attacks or, in other words, to design *survivable*

systems [13].

Motivated by the entity and damage of recent incidents, in this thesis we redesign the affected IT systems and components in a survivable way, to fix the fundamental design issues that allowed such incidents to occur. As cloud-based businesses have been the major target of the considered attacks, we start by considering the general design of an IT network in the context of a cloud-based business, and propose the first survivable zero trust architecture [50]. Then, given the central importance of authentication for zero trust security, we focus on a specific component of the proposed architecture, the survivable single sign-on component, to both fix limitations of existing survivable password-based SSO [90], and to provide the first passwordless survivable SSO protocol [89]. Finally, as the most damaging attacks have compromised the software supply chain of critical proprietary software adopted by thousands of organizations, we propose the first survivable update framework for proprietary software, to mitigate supply chain attacks [91].

As businesses have been the preferred attack target, we first focus on the general design of a survivable IT network in the context of a cloud-based business, by considering the emerging *zero trust* paradigm which seems to partially embrace survivable design guidelines. In fact, it recognizes that perimeter defenses can be breached and are no longer suitable for modern dynamic organizations. As a result, it suggests to consider users and their user agents as hostile, and to enforce access control on each request. However, existing zero trust architecture (ZTA) proposals enforce such access control by relying exclusively on trusted components. Thus, despite assuming adversaries capable of breaching perimeter defenses, they also assume that the same adversaries are not able to breach ZTA components. To overcome these unrealistic assumptions we propose the first survivable ZTA. The proposed architecture is able to tolerate intrusions and recover from failures and successful attacks [50].

One of the most critical parts of the proposed survivable ZTA is the survivable single sign-on (SSO) authentication component, which is responsible for password-based user authentication. Although survivable password-based SSO proposals exist, once deployed they fix the security parameters that allow to tune the amount of tolerable compromised components. This lack of flexibility makes the state-of-the art unsuitable to heterogeneous cloud environments that need to protect resources with diverse security requirements. We overcome this lack of flexibility by proposing the first flexible password-based SSO protocol [90] which allows to choose the trade-off between security and performance at run-time, and even to preserve compatibility with non-survivable SSO.

In the context of survivable authentication, passwords are among the most convenient credentials in terms of ease of use and user experience. However, all survivable password-based SSO proposals inherit the typical vulnerabilities related to password usage, including notoriously effective and arguably simple phishing attacks. Passwordless authentication provides phishing-resistance, but no existing passwordless protocol is designed to also guarantee intrusion tolerance. To fill this gap we propose SPOC, the first survivable passwordless SSO protocol [89]. SPOC preserves the usability of standard passwordless protocols

and is fully compatible with existing authenticators available on the market.

Finally, as the pervasive nature of IT has created a network of dependencies among businesses that produce software (software-houses) and possibly thousands businesses that consume said software, software supply chain attacks tend to have a large impact. Even though a single software-house can adopt the proposed survivable ZTA and survivable SSO protocols to defend against highly skilled adversaries, protecting the business software supply chain, composed third parties that are outside the business control, requires an ad-hoc solution. For this reason, to address supply chain attacks in the context of proprietary software distribution [92, 39], we introduce the first survivable software update framework for proprietary software [91]. Compared to previous works, the proposed framework is the first to guarantee both survivability, and confidentiality of software updates.

This thesis is structured as follows. In Chapter 2 we propose the survivable zero trust architecture. Chapter 3 describes the survivable and flexible password-based SSO protocol, while in Chapter 4 we show the survivable passwordless SSO protocol. In Chapter 5 we describe the proposed survivable software update framework. Finally, Chapter 6 concludes the thesis.

Chapter 2

Survivable Zero Trust for Cloud Computing Environments

2.1 Introduction

Cyber security based on the defense of the perimeter of an organization is becoming obsolete and unsuitable to organizations that are characterized by "bring your own device" policies, hybrid cloud services, remote working and Internet-of-Things. As a further flaw, the implicit trust granted to internal hosts facilitates lateral movements to attackers who have gained unauthorized access to the inner network and cloud resources [11]. The emerging security model based on the so-called *zero trust* [70] removes the concept of perimeter by designing architectures that do not grant implicit trust to any host/device because of their physical location. Access to any resource must be previously authenticated and authorized and, to this aim, zero trust solutions adopt two types of architectural components: the control plane and the data plane [60, 104]. The former includes components that collaborate to evaluate access control policies and configure data plane components. The latter elements enforce access control policies to the resources according to the configuration received by the control plane.

As every network flow is considered untrusted, zero trust architectures assume omnipresent internal and external threats [70, 60, 36, 113]. However, existing designs of zero trust architectures in literature assume an untrusted data plane but a trusted control plane (e.g., [44, 77, 46, 115, 122]) that is, they assume that adversaries are confined to the data plane cannot compromise the control plane components.

We argue that relying on a trusted control plane is incoherent with zero trust principles as we consider unrealistic to assume that attackers could not compromise control plane components that grant access to system resources. For this reason, we identify possible threats and attacks to zero trust control plane and mitigate them through a novel design that distributes trust within the control plane so that no single component is fully trusted. We evaluate the feasibility of the proposal in cloud computing environments by analyzing performance of the state-of-the-art protocols and components that are required for the implementation of the proposed architecture.

This chapter is organized as follows. Section 2.2 discusses related work. Section 2.3 introduces the reference zero trust architecture. Section 2.4 highlights possible attack scenarios. Section 2.5 overviews the design of the proposed survivable zero trust architecture to prevent the considered attacks. Section 2.6 describes the adopted technical solutions. Section 2.7 discusses the security and robustness of the proposed architecture. Section 2.8 discusses the feasibility of the architecture. Section 2.9 concludes the chapter.

2.2 Related work

In this work, we propose the first survivable zero trust architecture for cloud computing that satisfies the core principles of zero trust [70, 60, 36, 113], namely: the network is assumed to be hostile and characterized by persistent external and insider threats; network flows are not trusted even if they originate from internal networks; each flow must be authenticated and authorized through dynamic access control policies. Existing literature on zero trust assumes that only the data plane can be hostile while the control plane is assumed as trusted (e.g., [44, 77, 46, 115, 122]). As this assumption contrasts zero trust core principles, we design an architecture that tolerates attacks from external and insider adversaries against the data plane and the control plane.

Some works [44, 46] propose zero trust architectures composed by multiple trusted control plane components that do not guarantee security in the case of successful intrusions. These proposals do not store access control policies in a survivable way, hence an adversary which alters them may defeat access control mechanisms. We avoid a similar limitation by storing access control policies in survivable databases that tolerate attacks against policy integrity. As a further limitation, existing proposals assume trusted communication channels among control plane components, hence they cannot guarantee authenticity of control plane configurations. The proposed zero trust architecture secures control plane communications over untrusted channels with well-known public key infrastructure (PKI) and tolerates attacks against PKI components.

Another line of research (e.g., [77, 115] extend XACML [2]) proposes zero trust architectures supporting fuzzy evaluation of dynamic access control policies. These proposals rely on trusted components to evaluate policies and, as a result, they cannot tolerate attacks against the integrity of the policy evaluation process. We present an architecture supporting dynamic access control policies that can even tolerate integrity violations of the policy evaluation process.

The paper in [122] proposes a zero trust architecture that allows perimeterization of microservices, but a similar scenario is quite different than that considered in this work. We refer to users requesting access to resources through managed devices, whereas the other work considers microservices requiring access to other microservices. The difference between the two scenarios is that the entity issuing a request, known as *network agent* [60], in our case is represented by the pair user-device, whereas in [122] refers to microservice identities. This difference is reflected in the different design of the architectural components devoted to network agent authentication. In their proposal, the trusted kernel of the host executing a microservice authenticates requests originating

from the microservice. As a result, compromising the host allows the adversary to impersonate any microservice. The architecture proposed in this work authenticates user and device through intrusion tolerant components that prevent impersonation of a network agent by removing single points of vulnerability.

Recent literature aims to reduce trust assumptions on the control plane by proposing access control architectures based on blockchain [86, 125, 88, 80, 45]. For example, the authors in [45] extend the seminal zero trust architecture of [70] by storing and evaluating access control policies on smart contracts. The works of [86, 125, 88] extends XACML architecture by designing policy enforcement and decision points as blockchain components. These design choices guarantee the integrity of policies and of the evaluation process even in presence of attacks, but they incur in monetary and computational costs associated with permissionless blockchains. We propose a zero trust architecture with the same security guarantees without incurring in their drawbacks.

The authors in [80] propose an access control system in Industry 4.0 by adopting a permissioned blockchain that saves monetary costs, but this system considers network agents consisting of devices only and does not include user identities. As a result, it lacks the procedures and components required for user authentication, such as a user database or a single sign-on authentication system. This proposal does not satisfy the zero trust paradigm that both users and devices must be authenticated. Our architecture includes survivable components dedicated to user and device authentication, thus embracing and enhancing zero trust principles completely.

2.3 Reference zero trust architecture

Multiple architectures have been proposed to instantiate zero trust design principles at different levels of abstraction and coverage of security requirements. We consider a reference architecture that is a subset of the BeyondCorp Zero Trust Architecture (ZTA) [118], which is a full-fledged architecture already deployed in production-ready cloud-based environments. In Section 2.3.1, we show that the architecture has general validity and we discuss how it relates to NIST ZTA [104] and XACML architectures [2, 12].

The reference model of the considered ZTA is outlined in Figure 2.1. We consider the following components.

- **Administrator**: an employee with the privileged access to the architectural components. He can allow their initial setup and subsequent configurations.
- **User**: a person that requests access to a resource through a managed device.
- **Resource**: applications, services and infrastructures that are subject to access control.
- **Managed device**: device that a user adopts to access a resource; this device is procured and managed by administrators through some device management systems.

A similar architecture requires several components that are responsible for authentication and authorization as reported below.

- **Single Sign-On**: a centralized user authentication service that issues authentication tokens after validating users credentials.
- **Access Control Engine**: a centralized logical component that evaluates access policies and performs authorization by granting or revoking access to a resource.
- **Access Proxy**: a logical component that allows or denies access to resources as instructed by the Access Control Engine.

The Access Control Engine performs authorization by executing a so called *trust algorithm* [104] that evaluates access control policies by taking into account several information. It supports the evaluation of the trust level associated to a device-user pair requesting a resource. The sources of information, that are collectively referred as *Sources*, are:

- **Access Policies database**: stores the set of both infrastructure-wide and application-specific policies. A policy is a set of predicates that must be satisfied to access a resource.
- **Device Inventory database**: stores information about devices; the stored information includes device last observed status (e.g., operating system version or installed software) as well as other additional information (e.g., device owner or device certificates).
- **Device Certification Authority**: issues certificates to managed devices, which are stored in the Device Inventory as well as on devices.
- **Infrastructure Certification Authority**: issues certificates to other ZTA components.
- **User/Group database**: stores information about users and groups. This information is used to evaluate access control policies.

The third role is represented by the **Human Resources Staff** that is granted to write permissions by Administrators over the User/Group database and the Device Inventory.

Administrators grant write permissions to Human Resources staff by defining appropriate access policies in the Access Policies database. Human Resources staff is the only unprivileged actor that is allowed to access the Device Certification Authority, the Device Inventory and the Users/Group databases. A similar grant is denied to normal users.

We highlight that Users, Human Resources Staff and Administrators correspond to the typical roles of an organization. This classification reflects the different levels of trust required to access components and to operate the infrastructure, but it does not limit the general value of the proposal. We can model

all personnel that requires access to resources and that must not have access to the User/Group database, the Device Inventory or certification authorities with the User role. Moreover, the considered system model allows personnel to be assigned multiple roles at once (e.g., an administrator that also provisions managed devices).

Let us describe the flow of operations of the reference ZTA. When a new user needs access to resources, the Human Resources staff must complete the *user registration* and *device provisioning* operations. *User registration* is performed by authorized Human Resources staff which adds the new user to the User/Group database. Human Resources staff then completes *device provisioning* by obtaining a new device certificate from the Device Certification Authority, and by inserting a record in the Device Inventory database that contains device information and the device certificate. Once users and their managed devices are registered, they can obtain access to a resource by completing the authentication and authorization flow which involves the following sequence of operations: *device authentication*, *user authentication*, *request authorization* and *resource serving*. A request for a resource is always received by the Access Proxy which is in charge of *device authentication*. During device authentication the device identifies itself to the Access proxy by presenting the certificate. If the access proxy verifies that the supplied certificate is present in the authentication procedure, then it completes the authentication procedure by executing a public key authentication protocol with the device. If not, the Access proxy terminates the device authentication procedure. Once the device is authenticated, the Access Proxy redirects the user to the single sign-on system for *user authentication*. The user authenticates to the single sign-on system, possibly through multi-factor authentication, and obtains an authentication token attesting the user identity. The single sign-on system redirects the user back to the Access Proxy which forwards the user request, along with the obtained authentication information, to the Access Control Engine for *request authorization*. The Access Control Engine executes the trust algorithm by evaluating the request, the attached authentication information, as well as other data sources, and then determines whether access to the resource is granted or denied. The Access Control Engine finally sends the authorization result to the Access Proxy. If the access is granted, then the Access Proxy proceeds to *resource serving* during which the Access Proxy serves the resource to the user by acting as a reverse proxy. If the access is denied, then the Access Proxy terminates the sequence of operations.

All architectural components communicate over mutually authenticated and confidential channels. Channel authenticity relies on a public key infrastructure (PKI) whose root of trust is represented by the Infrastructure Certification Authority that is managed and maintained by administrators.

The Device Inventory database plays an important role during the device authentication procedure, because it acts as a certificate whitelist that allows only registered certificates to be used during authentication. This feature has relevant security implications that are discussed in Section 2.4 and Section 2.5.

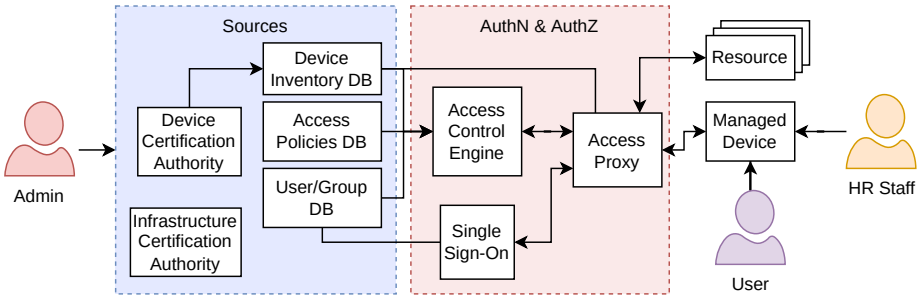

Figure 2.1: Reference zero trust architecture

2.3.1 Discussion on related architectures

We discuss how the considered architecture relates to BeyondCorp ZTA [118], NIST ZTA [104] and XACML architectures [2, 12].

The considered reference architecture is a subset of BeyondCorp ZTA (as already mentioned at the beginning of Section 2.3), because it focuses on preventive security measures, such as access control components that, if breached, give immediate unauthorized access to resources. BeyondCorp ZTA also includes additional supporting components such as security information and event management (SIEM) or threat intelligence systems. Our proposal is orthogonal to these supporting components and can be easily integrated with existing systems.

The NIST ZTA can be regarded as a full-fledged architecture that is similar to the BeyondCorp ZTA, however it adopts a higher abstraction level. The considered architecture represents a potential instantiation of the NIST ZTA: the Access Control Engine instantiates the Policy Decision Point (PDP), the Access Proxy instantiates the Policy Enforcement Point (PEP), the Access Policies database instantiates the Policy Administration Point (PAP), the User Group database instantiates the Subject database, the Device Inventory database instantiates the Asset database, the Single Sign-on instantiates the ID Management System, and the Device and Infrastructure certification authorities instantiate the Enterprise PKI component. Finally, similarly to how our architecture is a subset of the BeyondCorp ZTA, it would also be a subset of the NIST ZTA because it only considers preventive security measures. Other supporting components could be easily integrated with our proposal.

Standard XACML architectures [2] focus on access control components but do not include many details regarding authentication. For access control, XACML architectures broadly share the same high level abstraction approach of the NIST ZTA, and many components also have the same names (e.g., PEP, PDP, PAP). Thus, our architecture can be also considered an instantiation of these components of XACML architectures. However, some components are even more abstract because they do not consider details related to distinguished authentication of users and devices (as an example, the Policy Information Point (PIP)

store information that all ZTAs store in separated databases, such as the Subject database and the Asset inventory databases of the NIST ZTA). Finally, standard XACML architectures do not include other components, such as the Device Certification Authority and Single Sign-On system.

Some proposals extend XACML architectures to include identity management components to support credential-based access control [12]. However, they consider a network agent that only includes the user identity and not his device, and thus lack the components devoted to device authentication.

2.4 Threat model and attack scenarios

We consider an attacker that wants to gain unauthorized access to some resource of an organization, such as users' information stored in databases or signing keys. The attacker may be an external threat, who does not possess a device registered in the Device Inventory and who is not registered in the User/Group database. The attacker may also be an insider threat, such as a disgruntled employee, who owns a managed device registered in the Device Inventory and who is registered in the User/Group database. Our proposal can defend against attackers that try violating Sources as well as any authentication and authorization component in order to compromise the integrity and authenticity of the access control process.

Attacks that gain access to Sources and other authentication and authorization components by violating the credentials of administrators or employees do not qualify as a component compromise because the attacker uses valid credentials. To prevent these attacks, our proposal can be combined with standard solutions for multi-factor authentication or decentralized governance approaches proposed by the literature [123].

An attacker may try subverting the correct operations of zero trust access control by compromising the several components that, in the reference design of ZTAs, represent single points of vulnerability:

- **Device Certification Authority**: the attacker may compromise the confidentiality of the Device Certification Authority signing key. This allows the attacker to issue arbitrary certificates to attacker-controlled devices ($A1$). As the Device Inventory acts as a certificate whitelist, this attack does not allow the attacker to successfully authenticate his device because a corresponding entry that associates the rogue certificate to the attacker device must be present in the Device Inventory.

- **Infrastructure Certification Authority**: the attacker may compromise the confidentiality of the Infrastructure Certification Authority signing key. A similar compromise enables the attacker to issue rogue certificates to impersonate architectural components ($A2$). This attack becomes very effective in subverting zero trust access control if the attacker impersonates the Access Control Engine to the Access Proxy. In this case, the attacker can bypass the authorization logic by providing fake authorization results to the Access Proxy.

- **Device Inventory**: the attacker may compromise the integrity of the

inventory by inserting arbitrary device entries ($A3$). This allows the attacker to insert an attacker-controlled device among the set of legitimate managed devices. If a similar operation is executed after the compromise of the Certification Authority, it allows the attacker to authenticate arbitrary devices ($A1 + A3$).

- **Access Policy Database**: the attacker may compromise the integrity of the access policies ($A4$), and grant access to unauthorized devices and users. If the attacker is an insider threat, this attack is sufficient to grant access to unauthorized resources as the insider device and user account are already registered in their respective databases. If the attacker is external, he can modify the policies to the extent that anonymous accesses are allowed.

- **Access Control Engine**: the attacker may compromise the integrity of the access control logic ($A5$) and alter the authorization logic to allow access to unauthorized devices and users.

- **Single Sign-On**: the attacker may compromise the confidentiality of the single sign-on signing key ($A6$). This allows the attacker to sign arbitrary authentication tokens, thereby impersonating registered users. If an insider compromises the Device Inventory to assign an attacker-controlled device to the impersonated user ($A3 + A6$), the attacker can access unauthorized resources. An external attacker must also compromise the certification authority to issue a valid certificate to the attacker-controlled device ($A1 + A3 + A6$).

- **User Group Database**: the attacker may compromise the integrity of the user group database ($A7$) to grant access to unauthorized users. If the attacker is an insider threat, this attack is sufficient to grant unauthorized access to resources as the insider may change his attributes to elevate his access privileges. An external attacker may leverage this attack to register valid credentials to the Single Sign-On, but he must still compromise the certification authority and the device inventory to issue a valid certificate to the attacker-controlled device ($A1 + A3 + A7$)

- **Access Proxy**: the attacker may violate the confidentiality of the secret keys used by the Access Proxy to authenticate to resources ($A8$). Since the Access Proxy acts as a policy enforcement point, an attacker that is able to impersonate the Access Proxy can bypass any authentication and authorization logic.

We summarize the attacks against individual components and the cyber kill chains that allow unauthorized resource access in Table 2.1.

The presented attacks allow to gain unauthorized access to cloud resources through different numbers and types of additional components that must be compromised. For example, compromising the Infrastructure Certification Authority ($A2$), the Access Policy database ($A4$), Access Control Engine ($A5$), the

Attack label	**Attacked component(s)**	**Attack description**	**Unauthorized access**
$A1$	Device CA	key confidentiality	✗
$A2$	Infrastructure CA	key confidentiality	external
$A3$	Device Inventory database	data integrity	✗
$A4$	Access Policy database	data integrity	external
$A5$	Access Control Engine	process integrity	external
$A6$	Single Sign-On	key confidentiality	✗
$A7$	User Group database	data integrity	insider
$A8$	Access Proxy	key confidentiality	external
$A3 + A6$	Device Inventory database Single Sign-On	kill chain	insider
$A1 + A3 + A6$	Certification Authority Device Inventory database Single Sign-On	kill chain	external
$A1 + A3 + A7$	Certification Authority Device Inventory database User Group database	kill chain	external

✗: attack does not grant unauthorized access

Table 2.1: ZTA attacks and kill chains summary

User Group database ($A7$) or the Access Proxy ($A8$) allows unauthorized access to resources because the attacker is able to alter or bypass the authorization logic without the need of compromising additional components. As a result, these attacks are very effective and require a minimum amount of successful compromises. We note that attacks $A1$, $A3$ and $A6$ do not grant unauthorized access on their own. In order to be effective, they must be included in a cyber kill chain consisting of multiple attacks.

If the attacker is unable to alter or bypass the authorization logic, then he must carry out attacks requiring the compromise of multiple components. In a similar way, the attacker can issue a request from an authorized device while impersonating an authorized user.

It is tempting to assume that compromising multiple architectural components is harder than compromising a single component and, as a consequence, that attacks involving the Device Certification Authority, the Single Sign-On system, the User Group database and the Device Inventory ($A3 + A6$, $A1 + A3 + A7$, $A1 + A3 + A6$) are unfeasible or unrealistic. This assumption is false when the architectural components are affected by the same vulnerabilities. For example, when they rely on the same software dependencies (e.g., same libraries), if they share the same operating system or if they run on the same hardware [81, 72]. In practice this is a common occurrence as the additional costs of n-version programming [14] and of supporting heterogeneous hardware inhibit the creation of failure independent subsystems. In these scenarios, the efforts of compromising multiple components may not be more expensive than compromising one of them. As a result, designing survivable ZTAs

for cloud computing environments that can resist to classes of seemingly complex attacks against multiple architectural components is of urgent and practical interest.

2.5 Zero trust architecture survivable design

The core zero trust principles [60] that guide our design can be summarized as following:

- the network is always assumed to be hostile;
- external and internal threats exist on the network at all times;
- accesses from the local network do not guarantee trust;
- every device, user, and network flow must be authenticated and authorized;

These core principles are established to protect network resources, but we claim that they should also be extended to protect the ZTA itself from external and internal threats. Indeed, the multitude of attacks undermining the security of architectures presented in Section 2.4 have the common characteristics of violating the trust of some components. The mitigation of the impact of trust violations due to successful compromise of trusted components requires the distribution of trust of each component among multiple entities. We identify two main approaches: *intra*-component trust distribution and *inter*-component trust distribution.

Intra-component trust distribution substitutes the original trusted component with a functionally equivalent structure composed by multiple replicas. Replicas share trust, so that the corruption of a subset of replicas is not sufficient to affect the security properties of the replicated component. Byzantine fault tolerant state machine replication is a well-known example of a similar approach [109].

Inter-component trust distribution requires the distribution of trust among heterogeneous components sharing some dependency relation. This solution allows components to share trust on their dependencies with other components or actors. Whitelisting is an example of inter-component trust distribution: a component considers data received by a dependency as valid if it is both correct and approved. Data is correct if it satisfies certain semantics and formal properties. It is approved if it finds a match in the local whitelist. This component distributes trust because it allows the dependent component to discard seemingly valid but not approved data produced by a malicious dependency. To this aim, whitelisting relies on a trusted entity to define and update the whitelist. As a result, trust is shared between the dependency and the trusted entity: the dependency is trusted for correctness, while the entity defining the whitelist is trusted for approval. In our proposal, administrators represent the trust anchor of the ZTA. We can rely on them for whitelisting operations. Inter-component trust distribution based on whitelisting is already present in the reference ZTA in the form of the Device Inventory. It acts as a device certificate whitelist which

prevents dependent components (e.g., Access Control Engine and Access Proxy) from accepting forged device certificates that are correct but not approved. We rely on this observation to avoid the application of intra-component trust distribution techniques on the Device Certification Authority and the Infrastructure Certification Authority (see Section 2.6.3).

A so called *intermediary removal* is another example of inter-component trust distribution. Here, trusted dependencies are removed by shifting their responsibilities to the dependent components. This allows the system to distribute trust among all the components that depend on the removed intermediary because each component is independently responsible for guaranteeing the properties and functionality offered by the removed intermediary. In our proposal, we apply this technique by designing protocols that allow to move access control enforcement responsibilities from the Access Proxy to resources. This allows us to avoid the application of intra-component trust distribution techniques on the Access Proxy.

To sustain the workloads of modern cloud computing environments, the scalability and elasticity properties offered by cloud infrastructures are essential. Therefore, we consider the architectural components of our design as part of a cloud infrastructure owned and maintained by the infrastructure operator. Admins preserve direct access to the cloud infrastructure and, in accordance with the zero trust paradigm, our architecture allows users to access cloud resources both from within operator premises and remotely, provided they adopt authorized managed devices.

Our design of a survivable ZTA is guided by the threat model presented in Section 2.4. For each attack, we identify the security properties that an attack may violate, and we derive the security requirements that the affected logical components must satisfy. Finally, for each affected component we identify the most appropriate trust distribution technique that can mitigate the identified attacks and protect the affected security properties. This approach preserves the logical components of the reference ZTA and their original functions, but it requires some additional components that operate together to guarantee the survivability of the architecture. The main requirements that each component must satisfy are outlined in Figure 2.2, while the resulting design is shown in Figure 2.3.

Some attacks are possible because conventional databases do not tolerate malicious violations of the data integrity. As this attribute must be one of the main goals of a survivable architecture, we require that the Device Inventory, the User/Group database and the Policy Database can tolerate integrity violations by design and can recover from attacks. To this aim, we apply intra-component trust distribution and replicate the Device Inventory, the User/Group database and the Policy Database through *Byzantine Fault Tolerant* (BFT) database replication. These schemes distribute a database among a set of replicas that can mask malicious integrity violations and recover from them.

The Device Certification Authority is a single point of vulnerability because it is not designed to tolerate confidentiality violations of its signing key. An attacker that compromises the Device Certification Authority can issue rogue

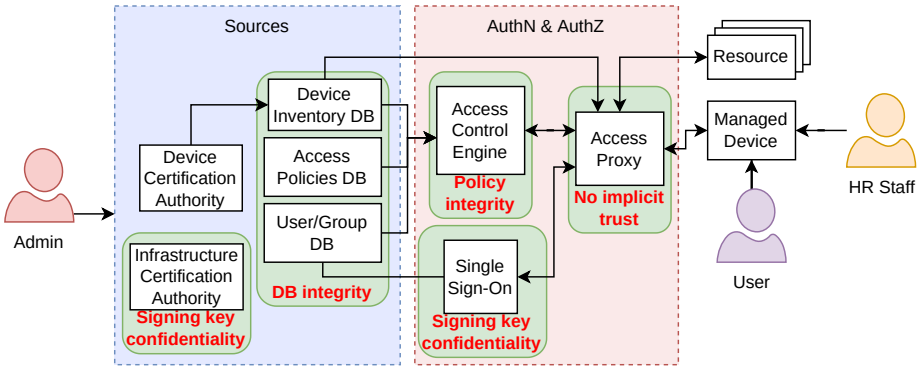

Figure 2.2: Security requirements that the components of a survivable zero trust architecture must satisfy

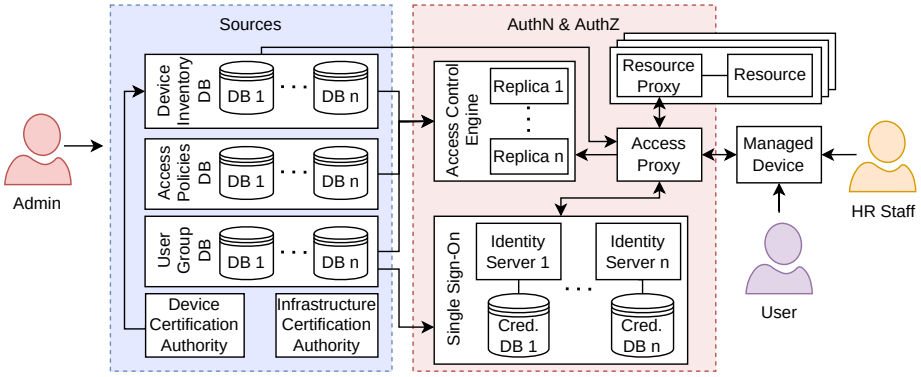

Figure 2.3: Overall survivable zero trust architecture

certificates to attacker-controlled devices that can later try to authenticate by presenting a valid certificate. However, a BFT Device Inventory is sufficient to prevent this attack. In fact, the Device Inventory acts as a certificate whitelist that prevents authentication with unregistered certificates. Since the attacker is unable to maliciously alter a BFT Device Inventory to insert rogue certificates, he is unable to authenticate his rogue devices. Our design leverages a similar whitelist-based inter-component trust distribution, and does not require a Device Certification Authority that can tolerate the violation of signing key confidentiality because the BFT Device Inventory already masks this attack.

The Infrastructure Certification Authority is a single point of vulnerability as an attacker that compromises its signing key can issue a rogue certificate to impersonate infrastructure components. In order to mitigate this attack, we apply whitelist-based trust distribution on the Infrastructure Certification

Authority that allows dependent components to detect forged certificates not belonging to the component local certificate whitelist. As a result, our design can securely adopt conventional certification authorities that rely on well-established security techniques to detect and recover from key compromise.

The type of user impersonation attacks violates the confidentiality of the signing key that is used to issue single sign-on authentication tokens to users. In such a way, an attacker is able to forge rogue authentication tokens. For this reason, we require single sign-on authentication mechanisms that can tolerate and recover from attacks that compromise the confidentiality of the signing key. To guarantee signing key confidentiality, we apply intra-component trust distribution by adopting a *survivable single sign-on* mechanism. This mechanism splits the signing key among multiple signing parties so that, if a subset of parties is compromised, then the confidentiality of the signing key is guaranteed; moreover, the compromised parties can recover from an attack. We allow each signing party to access the Users/Group database to include user information in the issued authentication tokens.

The Access Control Engine represents a single point of vulnerability because an adversary that can compromise the integrity of its access control logic may be able to gain unauthorized access to cloud resources. To guarantee integrity of the access control logic we apply intra-component trust distribution by replicating the Access Control Engine through BFT schemes so that it can tolerate the compromise of a threshold of its replicas.

Finally, the Access Proxy is another single point of vulnerability because an adversary breaching it can violate the confidentiality of its keys used to establish authenticated communication channels with cloud resources. This vulnerability allows the attacker to access resources by impersonating the Access Proxy and bypass any authentication and authorization logic. This is due to the implicit trust that each resource grants to the Access Proxy. To fix a similar vulnerability we notice that the Access Proxy has many responsibilities that do not have to remain concentrated in a single component. In particular, the Access Proxy is responsible for both device authentication and policy enforcement. Device authentication can be left to the Access Proxy, while policy enforcement can be decentralized. To this aim, we apply inter-component trust distribution based on intermediary removal by extending each resource with a Resource Proxy, also known as *Zero Trust reverse proxy* [60] or *gateway* [104]. Each resource trusts its Resource Proxy to enforce access control. This design approach distributes trust among multiple Resource Proxies instead of concentrating it in a single Access Proxy. The advantage is twofold: a compromised Access Proxy does not allow an adversary to access resources; a compromised Resource Proxy does not allow an adversary to access other resources managed by other non-compromised Resource Proxies, thus limiting the impact of the compromise of a trusted component.

2.6 Detailed architecture

We describe how the trust distribution techniques of the architectural design are applied to each component. Section 2.6.1 analyzes survivable databases

used in multiple components of the architecture. Section 2.6.2 discusses the survivable single sign-on component. Section 2.6.3 discusses trust distribution for certification authorities. Section 2.6.4 describes the novel design of the access control engine and of the access proxy.

2.6.1 Survivable databases

Survivable databases can be obtained by means of middleware-based *byzantine fault tolerant (BFT) database replication* (also, BFT databases), which are full-fledged database management systems (DBMS) that achieve BFT guarantees. Although the initial BFT databases date back to some decades ago [57], they have been improved by more recent research proposals (e.g., [58, 59, 114, 56, 100, 84]). We show that no existing BFT database can satisfy all the requirements of our survivable ZTA.

Table 2.2 summarizes the analysis of existing BFT databases. In this table, we consider typical features [83], such as *number of replicas*, *message complexity*, *consistency guarantees*, support of *concurrent transactions* and *tolerance of faulty clients*. Moreover, we introduce additional requirements that are inherited by the literature on survivable systems [13]: presence of *single points of vulnerability* and *fault recovery*.

	SPoV	# replicas	Fault recovery	Message complexity	Consistency	Concurrent transactions	Faulty clients
HRDB	yes	$2k+1$	yes	$O(n)$	serializable	yes	yes
Gashi et al.	no	$3k+1$	yes	$O(n^2)$	serializable	no	yes
BFT-DU	no	$3k+1$	no	$O(n^2)$	serializable	yes	no
Byzantium	no	$3k+1$	no	$O(n^2)$	snapshot	yes	yes
MITRA	no	$3k+1$	no	$O(n^2)$	serializable	yes	yes

Table 2.2: Comparison among BFT database replication proposals

Single points of vulnerability. BFT database replication solutions may introduce single points of vulnerability (denoted as "SPoV" in Table 2.2) in their architecture as a compromise between survivability and performance.

Number of replicas. The number of replicas that a BFT database requires corresponds to the number of distinct database instances that must be deployed to guarantee byzantine fault tolerance. Most proposals require a number of $n = 3k + 1$ replicas, where k is the number of faulty replicas that the system can tolerate. For example, we need four replicas to tolerate $k = 1$ faults.

Fault recovery. This property guarantees that the database can recover from a compromise of some of its replicas. Moreover, it ensures that the number of faulty replicas f does not exceed the amount established at system setup during the whole lifetime of the database.

Message complexity. The message complexity indicates the growth of the number of messages that are globally exchanged among replicas during the protocol execution with respect to the number of replicas. Most proposals require a quadratic number of messages in the number of replicas.

Consistency guarantees. The consistency guarantees indicate the tolerance of the replicated database to anomalies arising from the concurrent execution

of transactions. Single-copy serializability (denoted as "serializable" in Table 2.2) guarantees that the execution of concurrent transactions on a replicated database behaves as the sequential execution of the same transactions on a single copy of the database [23]. This solution represents the strongest consistency guarantee for replicated databases. On the other hand, snapshot isolation is a weaker consistency guarantee that may affect data integrity because it does not prevent some classes of anomalies [22]. Other proposals adopt a similar isolation level because, especially for read biased workloads, it offers a better transaction throughput. In our case, single-copy serializability is preferable to snapshot isolation because an attacker may exploit anomalies to maliciously corrupt data.

Concurrent transactions. A database may not support the execution of concurrent transactions, such as in the case of [58]. This is a limitation that may negatively impact performance in scenarios characterized by intensive database workloads.

Faulty clients. A BFT database tolerates byzantine clients if it does not allow a client to compromise the database consistency guarantees during transaction execution. As an example, a byzantine client may equivocate responses to different replicas during protocol execution to bring the system in an inconsistent state.

The evaluation criteria allow us to identify the requirements that a BFT database should satisfy in a survivable ZTA. A BFT database should not introduce single points of failure. It should ensure single-copy serializability and allow concurrent transactions. It should tolerate byzantine clients and guarantee fault recovery. Existing proposals only partially satisfy all these requirements. Hence, further research efforts are required in the field of BFT databases. We identify MITRA and HRDB as the solutions offering the best trade-offs among the examined BFT databases. MITRA satisfies most requirements but it does not include a fault recovery mechanism. Hence, if faults can be detected, then the database may be restored to a known safe checkpoint, but non-detectable byzantine faults may go unnoticed and accumulate over time among replicas so that the security threshold is exceeded [59]. HRDB supports fault recovery and offers better performance than MITRA (Section 2.8) at the cost of introducing a trusted component in its design. The trusted component has a much lower attack surface compared to database replicas as it can be implemented with a smaller code base [114]. The adoption of secure coding and operations best practices to protect the trusted component should guarantee that HRDB can be regarded as a practical proposal in the context of survivable ZTAs.

It is interesting to evaluate some architectural features of existing BFT databases that determine how they may be integrated in a survivable ZTA. A common approach to BFT databases is to design a middleware that enables crash-fault tolerant databases to collectively guarantee a BFT service. The middleware intercepts client requests and delivers them to the appropriate replicas, thereby hiding BFT replication to the client. This black box approach has several advantages: it does not require modifications of the database source code, which may not be available in case of proprietary databases. Moreover, it allows

the inclusion of several databases from different vendors in the deployment of the BFT system. This feature is important because it allows middleware solutions to support *design diversity* [14], which is an important security technique to mitigate common-mode failures.

In general, the architecture of a middleware-based BFT database consists of a client-side proxy component that is aware of each replica in the system. The proxy intercepts the client requests and appropriately forwards them to the replica-side proxy which then communicates with the replica's database management system (DBMS).

Our design replaces the User/Group and the Access Policies databases with a different BFT database, and may replace the Device Inventory to obtain stronger survivability guarantees (for a precise definition of survivability guarantees see Definition 2.7.1). To this aim, we must establish the desired level of tolerable byzantine replicas k for each BFT database, and then deploy the required total number of replicas. Moreover, the single sign-on system, the Access Control Engine and the Access Proxy, which are clients of the BFT databases, must be modified to include the client-side proxy.

2.6.2 Survivable single sign-on

A survivable single sign-on function requires a cloud-based *survivable single sign-on* architecture. The examples in [5, 17, 123] are some single sign-on systems that guarantee signing key confidentiality in case of successful attacks and allow recovery from successful compromises. We establish the most appropriate system that our architecture can adopt, by analyzing the state-of-the-art about traditional and survivable single sign-on solutions and identify the characteristics that are required for survivable ZTAs.

Traditional single sign-on, represented in Figure 2.4, involves three actors: a user, and identity provider and a service provider. The user wants to access a resource owned by the service provider, who enforces access control over the resource. As a result, the service provider needs to authenticate users. To this aim, the service provider trusts and delegates to the identity provider the authentication procedure. The service provider does not maintain the user credentials database required for authentication, which is maintained by the identity provider. A user, who has previously registered himself to the identity provider, can use his credentials to authenticate to the identity provider. Upon successful authentication, the identity provider digitally signs and sends an authentication token to the user. The user then forwards the authentication token to the service provider. The service provider verifies the token authenticity and, if authentic, can be convinced about the user identity.

Various attacks against traditional single sign-on systems motivate the need for survivable single sign-on. An attacker that can compromise the confidentiality of the identity provider signing key is able to forge arbitrary authentication tokens. Moreover, an attacker that can read the credentials database may be able to mount offline dictionary attacks. A survivable single sign-on system, represented in Figure 2.5, prevents these attacks by substituting the identity provider with a set of n identity servers, each maintaining independent creden-

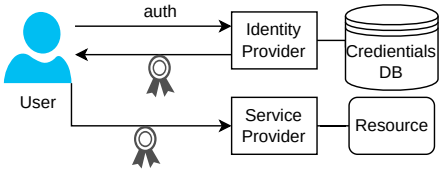

Figure 2.4: Traditional SSO architecture and flow

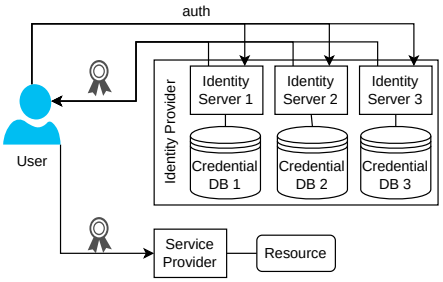

Figure 2.5: Survivable SSO architecture and flow

tials databases. The system can tolerate at most $k < n$ corrupt identity servers. To prevent an adversary from forging arbitrary authentication tokens, the identity provider signing key is cryptographically split among n identity servers so that any subset of less than $k + 1$ servers cannot reconstruct or use the signing key. This approach guarantees *token unforgeability* against the breach of k identity servers. To prevent an attacker from mounting offline dictionary attacks, user credentials are cryptographically split among multiple credentials databases so that gathering data stored in less than $k + 1$ databases does not reveal any information about user credentials. This guarantees *credentials security* against the breach of k credentials databases.

The information flow of survivable single sign-on is similar to the flow of traditional single sign-on. In survivable single sign-on, the user authenticates to a threshold of identity servers which issue enough partial authentication tokens so that the client can reconstruct a full authentication token. The client then forwards the authentication token to the service provider as in traditional single sign-on.

We identify the characteristics of survivable single sign-on (SSO) and establish the requirements that survivable SSO must satisfy to be adopted in a survivable ZTA. To this aim, we consider typical features adopted in the survivable single sign-on research field [17], that are: *threshold security*, *proactive recovery* support, *trusted setup*, number of *round trips for authentication*, resistance against *online password testing attacks*, *number of secret keys per replica*. Table 2.3 compares the considered proposals.

	threshold	proactive	Trusted setup	# auth. round trips	OPTA	# secrets per replica
PASTA	(k, n)	no	no	1	no	$O(\#users)$
PESTO	$(n-1, n)$	yes	yes	2	yes	$O(n)$
PROTECT	(k, n)	yes	no	2	yes	$O(\#users)$

Table 2.3: Comparison among survivable single sign-on proposals

Threshold security. This criterion indicates whether the SSO system allows

the choice of any threshold of tolerable malicious identity servers $k < n$ or imposes additional constraints on the value of k. In the case of PESTO [17], the value of k can be set just to $k = n - 1$. This implies that a token generated by this proposal is valid only if it is authenticated by all identity servers. Therefore, if a single identity server becomes unresponsive, the whole SSO becomes unavailable as no valid authentication tokens can be generated. As a result, allowing the flexible definition of the amount of malicious identity servers k, such as in solutions that guarantee (k, n) threshold security, can guarantee a higher level of availability of the single sign-on service than solutions that guarantee $(n - 1, n)$ threshold security. We note that all considered proposals [5, 17, 123] require to establish the values of k and n at protocol setup and do not allow to change them at runtime. In Chapter 3 we overcome these limitations and introduce a survivable SSO protocol that allows the service provider to choose the value of k at runtime according to the security requirements of the resource being accessed.

Proactive security. A survivable SSO proposal is proactively secure if it allows to periodically refresh the cryptographic material of identity servers. This allows us to recover compromised identity servers from successful attacks and ensures that any leaked cryptographic material becomes useless after a refresh. This property is essential to guarantee that the amount of compromised identity servers never exceeds the security threshold k during the whole lifetime of the single sign-on service.

Trusted setup. A survivable SSO may require a trusted setup to split cryptographic material among identity servers. A trusted setup introduces a possible single point of failure in the distributed architecture. However, in an enterprise scenario a trusted setup may be acceptable in case it is executed by an administrator who must be trusted anyway because, for example, he needs direct access the identity servers for setup and maintenance purposes. Moreover, schemes that rely on trusted setup usually are more efficient than alternative schemes not relying on trusted setup. As an example, PESTO, which relies on a trusted setup, allows a non-interactive proactive refresh phase. PROTECT, which does not rely on a trusted setup, incurs in a quadratic communication cost among servers during the proactive refresh procedure.

Number of round trips for authentication. It indicates the number of round trips of communication between a client and each identity server during authentication. Current proposals either require one or two round trips of communication. While reducing the number of round trips from two to one clearly benefits communication efficiency, research shows that this opens the possibility of online password testing attacks (OPTA) [17, 123], which we describe shortly. Determining whether it is possible to reduce the number of rounds to one while preventing OPTAs is still an open problem.

Online password testing attacks. This criterion indicates whether an attacker is able to perform an online password testing attack. This attack allows an adversary with a dictionary of candidate passwords to retrieve the identity of users using a particular password in the dictionary. A desirable property of a survivable single sign-on system is to prevent these attacks as this allows weaker

assumptions on the strength of user credentials.

Number of secrets per replica. It indicates the growth of the amount of secret cryptographic material as a function of some parameters, such as the number n of identity servers or the number of users (denoted as "$\#users$" in Table 2.3). In PESTO, user-specific information stored in each credential database is not confidential and as a result, the amount of confidential cryptographic material does not depend on the number of users that we assume to be much larger than the number of identity servers. On the other hand, PASTA and PROTECT register users by storing security-sensitive cryptographic material in the credentials database and thus the amount of confidential cryptographic material depends on the number of users. A low amount of confidential cryptographic material may allow the identity servers to securely store such material in a hardware security module enhancing the overall security.

Let us identify the main requirements that a survivable SSO system must satisfy to be adopted in a survivable ZTA. First of all, it must guarantee proactive security and prevent online password testing attacks. Moreover, it must guarantee (k, n) threshold security to provide higher availability with respect to $(n - 1, n)$ threshold security. To the best of our knowledge, PROTECT is the only proposal satisfying all previous requirements.

To integrate the survivable SSO in our design, we assume that the Users/Group is a middleware BFT database. The survivable SSO system must have access to the Users/Group database to include user identity information in authentication tokens. This implies that each identity server must be modified to include the client-side proxy, as discussed in Section 2.6.1. There are different design decisions that may be considered as appealing optimizations, but some of them are useless or even harmful in terms of security. The two not recommended optimizations include the replication of the credentials databases of each identity server, and the memorization of all user credentials in one database. One may hope that replicating the credentials database of each identity server through BFT database replication can increase the security of the SSO system, but this is not true. Credentials database distribution in the context of survivable SSO enhances security by cryptographically splitting credentials at multiple locations to prevent offline dictionary attacks. On the other hand, BFT replication in the context of databases enhances data integrity and availability. Moreover, it prevents equivocation in presence of byzantine adversaries. These two security goals are distinct, hence replicating a credentials database does not increase the difficulty of offline dictionary attacks. In summary, replicating credential databases is useless in terms of security, and possibly harmful in terms of performance due to the overhead of BFT replication.

The second not recommended optimization involves the reduction of the number of credentials databases by storing user credentials in the Users/Group database. However, survivable SSO systems are designed with the assumption that credentials databases are distinct for each identity server and that breaches are independent. Storing credentials in the same database, even if it is BFT-replicated, violates this fundamental assumption as it allows an adversary who has breached the Users/Group database, or any of its replicas, to collect

all credentials at once, thus effectively violating more than k identity servers. Therefore, reducing the number of credentials databases is a bad design decision that harms the security of SSO and that must be avoided.

2.6.3 Certification Authorities

We detail the trust distribution techniques applied to the Device Certification Authority and the Infrastructure Certification Authority. of the reference ZTA. This latter CA is a single point of vulnerability that, if breached, allows an attacker to issue valid certificates to attacker-controlled devices and to impersonate other components of the ZTA. A possible solution is to continue the trend of intra-component trust distribution proposed in Section 2.6.1 and Section 2.6.2 towards the adoption of survivable certification authorities [103, 119, 124]. This is an unnecessary step because we show that applying inter-component trust distribution and maintaining conventional centralized certification authorities guarantees enough security against forged certificates.

Let us first consider the Device Certification Authority. We assume that the Device Inventory is BFT-replicated (Section 2.6.1) and, as a consequence, the attacker is unable to compromise its integrity. Hence, even if an attacker compromises the confidentiality of the Device Certification Authority signing key, the resulting rogue certificates issued to attacker-controlled devices are useless to the attacker because the Device Inventory acts as a certificate whitelist. Thus, even if an attacker issues a rogue certificate for a device that the attacker owns, the certificate does not allow the attacker to satisfy existing access control policies because the attacker device is not present in the Device Inventory. As a result, during device authentication (Section 2.3), the Access Proxy denies the authentication to the attacker device as its rogue certificate is not present in the Device Inventory. The only way for an attacker to effectively use a rogue certificate is to add it to the Device Inventory. As the Device Inventory is BFT replicated, the attacker cannot proceed in a similar way.

We now focus on the Infrastructure Certification Authority. The effect of attacks in which the adversary issues rogue certificates to impersonate architectural components can be mitigated by means of key pinning [41]. In fact, key pinning is a form of certificate whitelisting. In key pinning, administrators configure the several architectural components to communicate over secure mutually authenticated communication channels which adopt admin-installed certificates to authenticate the endpoints. As a result, even if the attacker issues a rogue certificate and uses it in an impersonation attempt, the attacked components refuse to connect to the attacker because the rogue certificate is different from the component's pinned certificate.

Our design adopts well-established security techniques to detect and recover from key compromise of centralized certification authorities. Moreover, our design relies on a scalable cloud infrastructure to ensure a high level of availability of the certification authority service to sustain the workloads of modern cloud computing environments.

2.6.4 Access Control Engine and Access Proxy

To mitigate attacks against a compromised Access Control Engine, we apply intra-component trust distribution to the Access Control Engine. Moreover, to mitigate the attacks against a compromised Access Proxy, we apply inter-component trust distribution based on intermediary removal, where we remove the responsibility of access control enforcement from the Access Proxy. To securely integrate the two design choices we deploy a secure communication protocol that guarantees end-to-end authenticity of user requests and Access Control Engine policy evaluations, and end-to-end confidentiality of resource responses.

We describe the details of our proposal by considering Figure 2.6. Our proposal extends resources with a *Resource Proxy*, which enforces access control policies on its corresponding resource. It accepts requests from the Access Proxy and forwards them to the resource only if they are collectively authenticated by a threshold of Access Control Engine replicas, thus preventing unauthorized access to the resource by a compromised Access Proxy. If we assume that an attacker cannot compromise more than a threshold of Access Control Engine replicas, then the attacker is unable to forge an Access Control Engine collective signature.

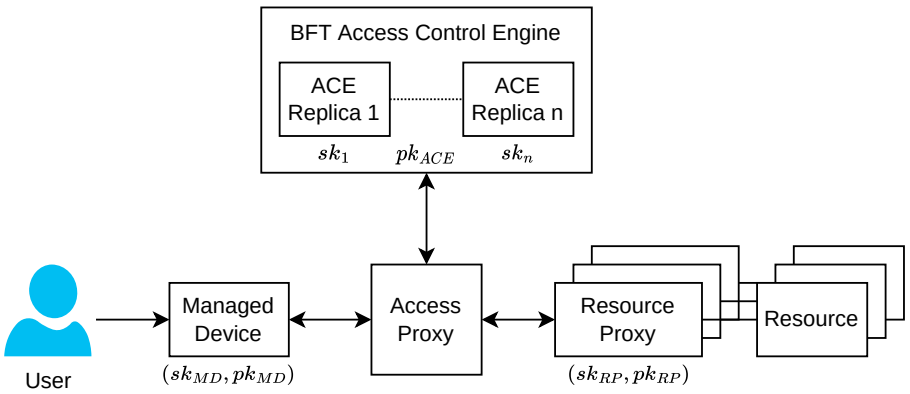

Figure 2.6: BFT Access Control Engine and Access Proxy architecture proposal

The high-level information flow of the proposed protocol is as follows. The user submits a request to the Access Proxy and completes both *device authentication* and *user authentication* as in the reference flow. If user and device authentication are successful, the Access Proxy sends the authenticated user request to the Access Control Engine replicas which evaluate the access policies. Replicas agree on the outcome of the policy evaluation and return a collective signature of the request to the Access Proxy if access is allowed. The Access Proxy forwards the signed request to the Resource Proxy which verifies the request signature and, if valid, the Resource Proxy forwards the request to the

resource. The resource sends the response to the Resource Proxy which encrypts it for the managed device and forwards the encrypted response to the Access Proxy. The Access Proxy then returns the encrypted response back to the managed device which decrypts it to obtain the plaintext response. We highlight that our proposal does not protect request confidentiality because it may have negative impact on system performance and, perhaps counterintuitively, even on security (we propose a discussion in the last paragraph of this section).

We now discuss how the proposed architecture can be deployed adopting existing solutions. The Access Control Engine can be decentralized by adopting *proactive byzantine fault tolerant* protocols, such as [40] by deploying n replicas. This protocol allows to achieve *byzantine fault tolerant state machine replication* [109] while also guaranteeing recovery from a compromise of $k < n/3$ replicas. Each replica of the Access Control Engine requires access to the Access Policies, Users/Group and Device Inventory databases to determine the outcome of the access control logic. Moreover, the Access Proxy requires access to the Device Inventory database to access whitelisted device certificates during device authentication. As a result, both Access Control Engine and Access Proxy replicas must integrate the client-side proxy of the adopted middleware-based BFT database replication solution discussed in Section 2.6.1. In our proposal, we assume that the setup of a proactively secure threshold signature scheme, such as [29], is completed so that each replica holds the secret share sk_i of the secret key corresponding to the replicas collective public key pk_{ACE}. The security threshold of the signature scheme must match the threshold k of the BFT protocol. We assume that the Resource Proxy knows the Access Control Engine collective public key pk_{ACE}. Moreover, we assume that the Resource Proxy owns a key pair (sk_{RP}, pk_{RP}) and that the managed device knows the public key pk_{RP}. In the following, we denote as (sk_{MD}, pk_{MD}) the public key pair of the user managed device, whose certificate is stored in the Device Inventory. Finally, we assume that communication among components occurs over confidential and mutually authenticated channels. We note that during proactive refresh, the cryptographic material of both the proactive BFT protocol and of the threshold signature scheme are refreshed.

We assume that the goal of the attacker is to gain unauthorized access to resources. To this aim, the attacker can compromise $k < n/3$ Access Control Engine replicas. The attacker can also compromise the Access Proxy to eavesdrop and modify all messages in transit and send new messages authenticated with the Access Proxy cryptographic material. Finally, we assume a computationally bound adversary that cannot break the security of the adopted cryptographic schemes.

We describe the detailed information flow and summarize it in Figure 2.7. The user generates the request, which we denote as req, to access a resource. The user managed device then computes the pair $\langle req, \sigma_{MD} \rangle$, where σ_{MD} denotes the signature on req produced with sk_{MD}. The user completes the user authentication and device authentication procedures (see Section 2.3) and sends $\langle req, \sigma_{MD} \rangle$ to the Access Proxy. We assume that req contains the due information required by the Access Control Engine BFT protocol, including metadata

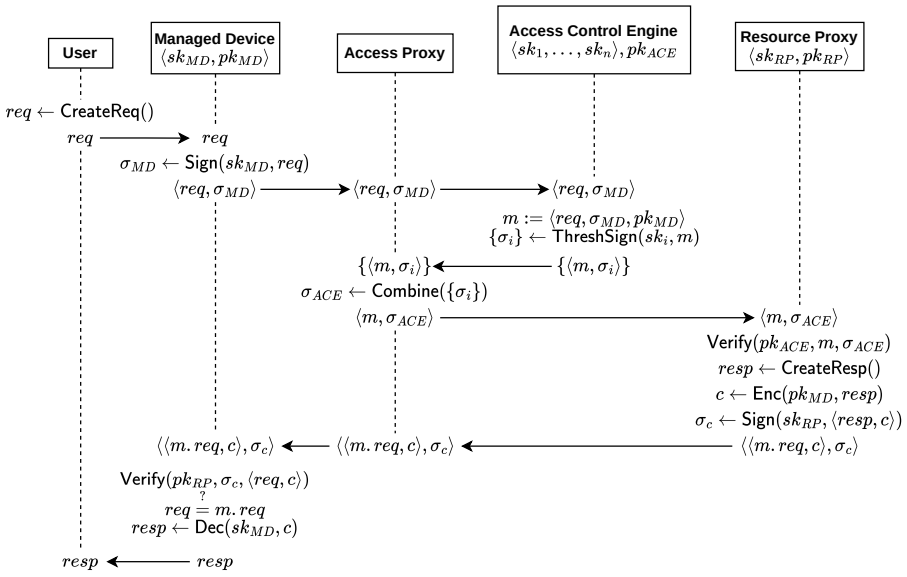

Figure 2.7: Resource access protocol flow

to defend against typical adversarial attacks (as an example, a nonce to prevent replay attacks). The Access Proxy sends $\langle req, \sigma_{MD} \rangle$ along with the authentication information obtained during device and user authentication to the Access Control Engine replicas. Each replica verifies the user and device authentication information as well as σ_{MD}. Each replica, given the authenticated request req, evaluates the access control policies. If access is granted, then replicas collectively sign an authorization message. To this aim, each replica computes $\langle \langle req, \sigma_{MD}, pk_{MD} \rangle, \sigma_i \rangle$ where σ_i denotes a signature share produced by replica i on message $\langle req, \sigma_{MD}, pk_{MD} \rangle$. We note that each replica has access to the Device Inventory so they can all obtain the same pk_{MD}. The Access Proxy collects at least $k + 1$ responses and then combines signature shares σ_i into a signature σ_{ACE} which can be verified with pk_{ACE}. The Access Proxy then sends $\langle \langle req, \sigma_{MD}, pk_{MD} \rangle, \sigma_{ACE} \rangle$ to the Resource Proxy. The Resource Proxy verifies σ_{ACE} with pk_{ACE} and, if the signature is valid, forwards req to the resource. The resource, given request req, computes the response $resp$ and then sends $resp$ back to the Resource Proxy. The Resource Proxy encrypts $resp$ with the managed device public key pk_{MD}, producing ciphertext c, and finally sings the pair $\langle req, c \rangle$ with his secret key sk_{RP}, producing signature σ_c. The Resource Proxy then sends $\langle \langle req, c \rangle, \sigma_c \rangle$ to the Access Proxy, which simply forwards it to the managed device. The managed device finally verifies that req matches the request sent initially, verifies σ_c with the Resource Proxy public key pk_{RP} and then decrypts c using his secret key sk_{MD}.

We now discuss the security of our proposal. The attacker is not able to alter

the request *req* sent by the user because it is authenticated by the user secret key, which is itself authenticated by the corresponding certificate stored in the Device Inventory. The attacker cannot replay the request to replicas because we assumed that *req* already embeds information to prevent replay attacks. The attacker is unable to forge an Access Control Engine collective signature as we assume that the attacker cannot compromise more than a threshold of replicas to gather enough key shares sk_i to produce arbitrary signatures. As a result, even if the attacker compromises the Access Proxy, it cannot gain direct access to resources. The attacker is unable to forge or replay old responses to the managed device because it is unable to forge Resource Proxy signatures which authenticate ciphertexts. Finally, the attacker is unable to eavesdrop on resource responses because they are encrypted for the managed device.

The availability guarantees of the proposed solution against denial of service (DoS) attacks involve some trade-offs that are worth discussing. We consider DoS attacks against the Access Proxy because it is a client-facing component and thus more likely to suffer from DoS. Moreover, we also consider DoS attacks originating from a compromised Access Proxy that silently drops user requests. In the proposed protocol Access Control Engine replicas collectively sign an authorization message only if access is granted. When access is denied to the user, Access Control Engine replicas do not respond with a collectively signed error message. The lack of response prevents the user from distinguishing an available ZTA that is denying access from an unavailable ZTA under DoS. To allow the user to detect DoS attacks, the protocol may be modified to also let Access Control Engine replicas collectively sign access-denied error messages. However, we note that collective signing is an expensive operation in terms of computational and network costs and thus signing every user request even when access is denied may facilitate DoS attacks because of the increased load on the Access Control Engine. Moreover, we note that the lack of response messages may actually be a desired behavior for a ZTA. In fact, Single Packet Authorization (SPA) is a technique that may be adopted to avoid fingerprinting of the infrastructure and resources. With SPA the Access Proxy only accepts ingress packets authenticated by known hosts, and silently drops packets from unknown hosts to conceal its presence [60]. If SPA is adopted, then revoked access at the SPA level is indistinguishable from an unavailable ZTA under DoS. Therefore, collective signing of access-denied error messages involves trade-offs that should be evaluated depending on the specific scenario that adopts the proposed architecture.

The security of the Resource Proxy component also deserves further consideration. In the proposed design the resource trusts its Resource Proxy for policy enforcement. Designing a survivable Resource Proxy would certainly enhance the security guarantees of the architecture. However, we conjecture that components responsible for access control *enforcement* must be trusted by dependent components (resources in our case). Our conjecture is supported by the fact that many related intrusion tolerant architectures independently obtain designs similar to ours by adopting trusted components for access control enforcement [55, 86, 87, 88, 45]. Designing a survivable enforcement point is an

interesting challenge for future work.

As anticipated at the beginning of this section, we note that the attacker is still able to eavesdrop on user requests. This may allow the attacker to infer enough information about the requested resource to compromise its confidentiality. However, we claim that protecting request confidentiality may have negative impact on system performance and on security. The confidentiality of the request must be protected at multiple points in the proposed architecture: it must be protected from a compromised the Access Proxy and it must be protected from any compromised Access Control Engine replica. To motivate the negative impact on system performance, we point out that to allow Access Control Engine replicas to evaluate policies while protecting request confidentiality, we must adopt *universally verifiable multiparty computation* schemes, such as [110]. These schemes allow a set of parties (the Access Control Engine replicas) to evaluate a function over secret inputs (provided by the user) and produce a result whose correctness can be publicly verified by the Resource Proxy which did not participate in the computation. These schemes are far from being practical and would impose a severe overhead on policy evaluation. Moreover, they have a negative impact on security because confidential requests prevent external tools from providing contextual threat information to the Access Control Engine, such as in the case of intrusion detection systems [11]. Therefore, guaranteeing confidentiality of requests and contextual request information may be conflicting requirements.

2.7 Security Analysis

In this section we analyze the overall security of the proposed survivable ZTA.

In the reference threat model we identify sequences of attacks that allow the adversary to gain unauthorized access to a resource. These attacks are also relevant to the proposed survivable ZTA. The difference is that attacks against a survivable component must compromise an amount of replicas that exceeds the component security threshold.

To quantitatively assess the effectiveness of our design choices we introduce novel definitions of survivability in the context of ZTAs.

Definition 2.7.1 (Zero trust survivability level)**.** *The survivability level of the zero trust architecture is the minimum amount of replicas that an attacker must compromise to gain unauthorized access to a resource in any attack sequence.*

Intuitively, we say that a ZTA is survivable if there is no attack sequence that allows an adversary to gain unauthorized access to a resource by compromising a single replica. We formalize this intuition in Definition 2.7.2.

Definition 2.7.2 (Survivable ZTA)**.** *A zero trust architecture is survivable if its survivability level is greater than one.*

Given Definition 2.7.1, Definition 2.7.2 and the attack sequences that apply to the proposed architecture, we can derive a meaningful criterion to assign values to the security threshold of each replicated component to determine the

amount of its replicas. We consider components that are not replicated as a single replica. We define the security thresholds of replicated components as follows:

- k_{DIdb}: device inventory database
- k_{APdb}: access policy database
- k_{UGdb}: user group database
- k_{ACE}: access control engine
- k_{SSO}: single sign-on

In Table 2.4 we report the costs of the identified attack sequences in terms of number of replicas that the attacker must compromise. We note that attack

Attack sequence	**Attacked component(s)**	**Attack cost**	**Attacker type**
$A1$	Device CA	-	-
$A2$	Infrastructure CA	✗	external
$A3$	Device Inventory database	-	-
$A4$	Access Policy database	$k_{APdb}+1$	external
$A5$	Access Control Engine	$k_{ACE}+1$	external
$A6$	Single Sign-On	-	-
$A7$	User Group database	$k_{UGdb}+1$	internal
$A8$	Access Proxy	✗	external
$A3+A6$	Device Inventory database Single Sign-On	$k_{DIdb}+k_{SSO}+2$	internal
$A1+A3+A6$	Certification Authority Device Inventory database Single Sign-On	$k_{DIdb}+k_{SSO}+3$	external
$A1+A3+A7$	Certification Authority Device Inventory database User Group database	$k_{DIdb}+k_{UGdb}+3$	external

✗: prevented attack
-: attack not effective even in reference ZTA

Table 2.4: Attack costs

sequences that include attacks $A1$ involve the compromise of the certification authority, which is not a replicated component and therefore counts as a single replica. We also note that the above costs are reported considering the best case scenario in which all components and replicas are failure independent. As already discussed in Section 2.4, it is essential to guarantee failure independence among components (inter-component independence) and within components replicas (intra-component independence). Otherwise, all components and replicas that share common mode failures count as a single replica in evaluating the complexity of attack sequences.

Attacks $A4$, $A5$, $A1+A3+A6$ and $A1+A3+A7$ allow an external adversary to access a resource, whereas attacks $A7$ and $A3+A6$ allows only an internal

adversary to gain unauthorized access to a resource. As shown in Section 2.6.4, the proposed solution prevents attack $A8$ because an adversary that compromises the Access Proxy is not able to gain unauthorized access to resources. Moreover, attack $A2$ is also prevented by certificate whitelisting.

The costs of the attacks highlight the importance of adopting a survivable Access Policy database, a survivable User Group database and survivable Access Control Engine. Adopting non-survivable alternatives ($k_{APdb} = 0$ and $k_{ACE} = 0$ respectively) does not allow to design a survivable ZTA that satisfies the proposed survivability definition. The costs of attack sequences $A1 + A3 + A6$, $A1 + A3 + A7$ and $A3 + A6$ highlight possible trade-offs in terms of component replication during a real deployment. It is sufficient to guarantee that the cost of these attack sequences is greater than one to obtain a survivable ZTA. As an example, to achieve the minimum survivability level it is sufficient to guarantee that the Device Inventory and Single Sign-On components are failure independent, while it is necessary to adopt a survivable Access Policy database, a survivable User Group database and a survivable Access Control Engine. To achieve higher survivability levels it is required to adopt the intra-component trust distribution techniques also for the Device Inventory and Single Sign-On.

To evaluate the security of the proposal it is also interesting to consider threats identified by NIST against ZTAs [104].

Subversion of ZTA decision process. The Subversion of ZTA Decision Process maps directly to attack $A5$ that alters the integrity of the decision process of the Access Control Engine. This attack is mitigated through BFT replication of the Access Control Engine.

Denial-of-Service or network disruption. The Denial-of-Service attack considered by NIST involves impeding availability of the Access Proxy and the Access Control Engine. The proposed solution mitigates unavailability of the Access Control Engine through BFT replication. Moreover, the Access Proxy can be geographically replicated with well-known techniques, thus mitigating unavailability.

Stolen credentials or insider threat. Insider threats or attackers that leverage stolen credentials are already considered and their impact mitigated. As discussed in Section 2.4, credential theft can be mitigated through multi-factor authentication. Moreover, stolen credentials are not sufficient because the adversary requires control of the user device. Finally, we consider insider threats and mitigate possible attacks that they can launch, such as $A7$ and $A3 + A6$.

Visibility on the network. NIST identifies encrypted traffic as a limitation in network visibility through traffic inspection. The proposal preserves network visibility by allowing inspection of user requests received by the Access Proxy which terminates confidential communication channels.

Storage of system and network information. Another identified threat involves gaining reconnaissance information by violating the confidentiality of Sources (e.g. Access Policies database) and supporting systems (e.g. SIEM). Access to security critical components must be restricted to administrators only. Moreover, the proposed architecture is secure even against adversaries that have complete knowledge of the architecture by minimizing single points of vulnera-

bility through survivable design practices.

Reliance on proprietary data formats or solutions. Lack of interoperability constitutes a relevant practical concern, especially in the context of survivable zero trust which relies on design diversity to mitigate common mode failures. Indeed, future challenges in the field of zero trust involve standardization efforts to guarantee interoperability and prevent vendor lock-in.

Use of non-person entities (NPE) in ZTA administration. Finally, the compromise of automated administration tools may allow attackers to perform unauthorized actions on the ZTA. As automated administration tools perform actions on behalf of administrators, they inherit their trust assumptions and thus it is essential to restrict access and protect them from abuse.

2.8 Feasibility of the architecture

In this section we analyze the performance of the proposal and discuss some limitations that could be addressed by future research.

2.8.1 Performance analysis

The proposal of an innovative architecture opens the important question of whether the overheads related to the survivable ZTA can be really applied to protect accesses to cloud computing environments.

We evaluate the overall response times of the two most expensive operations of the survivable ZTA from the user point of view (Section 2.3): *user authentication* and *request authorization*. We can anticipate that the proposed architecture is characterized by response times of less than two seconds for *user authentication* and in the order of some hundreds of milliseconds for *request authorization*. Hence, we can consider this architecture feasible from the performance point of view.

User authentication is based on the survivable SSO component. In Section 2.6.2, we identify PROTECT [123] as the best candidate for our survivable ZTA. Considering realistic scenarios where the threshold of tolerable malicious servers is set to one or two units, PROTECT offers authentication response times in the order of two seconds even for low-powered devices. This timing is considered acceptable even with non-survivable access control architectures, where the security of password-based authentication is based on the adoption of password-hashing operations that introduce significant delays [27, 66].

Request authorization is based on the survivable Access Control Engine and BFT databases. We first consider the performance of the survivable Access Control Engine, which relies on BFT state machine replication protocols, such as [71, 15, 55, 84]. These proposals analyze the protocols by adopting a security threshold of $k = 1$ that complies with our recommendations for the Access Control Engine in Section 2.7. For example, the authors in [71] implement an intrusion tolerant system by relying on the Prime protocol [9], which allows BFT state machine replication with strong upper bounds on response times. The authors evaluate performance in a local area network and show that requests can be processed with response times in the order of 35ms. The same Prime protocol is adopted by the intrusion tolerant system proposed in [15], where

performance evaluations show that the system processes 99.999% of requests within a response time in the order of 100ms in a wide area network. In [55] the authors adopt the MOD-SMaRt state machine replication protocol implemented through the BFT-SMaRt library [24] that can process requests with an average response time of less than 10ms in a local area network. The Prime protocol is the best candidate to implement the Access Control Engine as a replicated state machine because it guarantees upper bounds on response times even in wide area networks, which is relevant in cloud applications, while the MOD-SMaRt protocol does not guarantee response times upper bounds. To analyze the performance of BFT databases, we consider MITRA [84] that is based on the already mentioned BFT-SMaRt library) and HRDB [114]. In Section 2.6.1 we identified them as the best trade-off solutions among the examined BFT databases. These proposals evaluate the performance with security threshold set to $k = 1$, which also complies with our survivability recommendations for ZTA. Their response times are in the order of 200ms [114].

As expected, survivable SSO appears as the component responsible for the highest response times of the ZTA. When a similar authentication is required, the total response time of a user request is typically below 2 seconds because modern devices have characteristics much better than those used in PROTECT experiments. It is worth to note that an SSO authentication must be executed only when a user initiates a session. Further requests can leverage the valid token which can be verified with negligible overhead with respect to Access Control Engine and BFT databases response times [123]. After the initial SSO authentication, the response time of each user request remains in the order of hundreds of milliseconds or below.

2.8.2 Possible limitations

We now summarize the design trade-offs of the proposed architecture and highlight limitations that may be addressed by future work. Our architecture relies on survivable databases to securely store data even in presence of intrusions. Proposals such as MITRA [84] and HRDB [114] guarantee the best trade-offs in terms of security and performance among the examined databases (Section 2.6.1) and they are good candidates to obtain a real implementation. Each database has different assumptions about survivability: MITRA assumes a bounded amount of faulty replicas over the whole architecture lifetime; HRDB assumes a trusted component. These assumptions are acceptable trade-offs to guarantee survivability. Relaxing them deserves further research efforts in the field of survivable databases that go beyond the scope of this proposal.

A relevant design aspect is represented by the reduction of the amount of trust in the Access Proxy by separating policy enforcement in a distinct component that we identify as the Resource Proxy. This choice has several security benefits (Section 2.5) but requires each resource to trust the corresponding Resource Proxy for correct policy enforcement. Ideally, the Resource Proxy should be designed as a survivable component so that resources can rely on untrusted components for policy enforcement. While we are not aware of any theoretical limit to this extension, trusting enforcement points seems an intrinsic require-

ment of access control architectures.

It is worth to note that the proposed architecture adopts a novel protocol guaranteeing end-to-end confidentiality of the response sent by the resource to the user. The proposed version of the Access Control Engine requires plaintext requests to evaluate access control policies and thus the protocol does not guarantee end-to-end request confidentiality. Adopting cryptographic protocols that allow to evaluate policies on encrypted requests would add significant performance overhead and would prevent full operation of intrusion detection systems. The design of a feasible protocol for end-to-end confidentiality with limited negative effects on performance and operation is an open research challenge that can be considered for future work even for different application contexts.

2.9 Final remarks

Zero trust architectures are emerging as solutions that solve the limitations of cyber defenses based on traditional perimeter for the protection of modern organizations. Existing designs assume that cyber attackers are confined to the data plane enforcing access control policies and that they cannot compromise the control plane components establishing access control policies. We present a novel survivable zero trust architecture that overcomes this strong assumption. The proposed design distributes trust among multiple control plane components so that the architecture can tolerate intrusions and recover from successful attacks. The proposed architecture is appropriate whenever combining cyber resilience and security is an important requirement, as in the case of critical infrastructures. This proposal is feasible even from the performance point of view, but it is open to future research improvements in the field of databases satisfying survivability, and survivable protocols that can combine end-to-end confidentiality and access control policy evaluation.

Chapter 3

Flexible and Survivable Single Sign-On

3.1 Introduction

Single sign-on (SSO) is a popular protocol to authenticate users requiring access to multiple Web-based services. Typically, an identity provider manages one logical identity server that issues authentication tokens proving user identity to service providers. The centralized design of SSO protocols is vulnerable to authentication tokens forgery as demonstrated by recent incidents [39, 4] where cyber attackers compromised the identity server and forged tokens that falsely impersonated users towards service providers. As discussed in Section 2.6.2, this issue can be addressed by so-called *survivable SSO protocols* that can prevent user impersonation even in presence of attacks (e.g., [5, 17, 123]). In survivable SSO, the identity provider manages multiple identity servers. A user authenticates himself to a subset of identity servers that collectively sign an authentication token and demonstrate the user identity to service providers. The amount of signing identity servers must be greater than a security threshold which defines the maximum number of identity servers that the adversary can violate.

Existing proposals achieve survivable token release by signing tokens through threshold signatures, and survivable user authentication through distributed password-based authentication protocols [5, 17, 123]. The problem is that threshold signatures tend to be unrealistic in practice because they do not guarantee flexibility. They prevent service providers from dynamically adjusting the value of the security threshold during protocol execution and they are not backwards-compatible with existing SSO systems. The lack of flexibility and of backwards compatibility prevent service providers from offering services with different identity assurance levels [3, 63], and from the possibility of dynamically adjusting the threshold based on user contextual information as suggested by the recent zero trust paradigm [104].

We propose an original survivable SSO protocol where survivable token release is achieved by signing authentication tokens through conventional digital signatures instead of threshold signatures as in literature. This approach enables the design of a survivable token release scheme that guarantees flexibility and preserves backwards compatibility with non-survivable SSO solutions. More-

over, we show that it is possible to guarantee survivable user authentication through password-based protocols even if they are not designed for distributed architectures. We evaluate the security of the proposed token release scheme and show the security of the overall SSO protocol by considering existing password-based authentication methods.

The practical relevance of the proposal is twofold. First, flexible and survivable SSO can be leveraged to mitigate emerging attacks to access confidential cloud resources through rogue authentication tokens [39]. Moreover, the high security level that is guaranteed by flexible and survivable SSO protocols is a perfect combination for mission critical systems requiring robust and reliable authentication mechanisms even under attack [52].

The chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 describes the system model and the single sign-on framework. Section 3.4 describes the threat model. Section 3.5 discusses the details and guarantees of the proposed novel survivable token release scheme. Section 3.6 shows the security level of the proposal. Section 3.7 evaluates the security and flexibility of the proposal and of related SSO protocols when integrated with different credential management systems. Section 3.8 highlights final remarks.

3.2 Related Work

This work is related to recent results investigating SSO protocols with different trade-offs between survivable security guarantees and flexible configuration [5, 17, 123]. For example, the authors in [5] propose a SSO protocol that tolerates the violation of up to a threshold of identity servers. The value of the threshold can be configured at setup time to tolerate from one compromised identity server up to a dishonest majority including a single honest identity server. The problem is that this protocol does not guarantee recoverability because it does not include the due procedures to recover compromised identity servers to a safe state. As a result, the protocol cannot be considered survivable because recoverability is a mandatory security guarantee in these systems [13]. We propose a protocol that guarantees survivability by defining specific procedures to recover compromised identity servers.

The guarantee of survivability is also analyzed by the proposal of a password-based survivable SSO protocol [17]. The authors consider a strong adversarial model that requires to trade token unforgeability for availability, as the protocol does not terminate if a single identity server is unavailable. We assume a different model called *mobile adversarial model* [65]. Although it is weaker than that considered in [17], the mobile adversary is a realistic model for SSO scenarios and allows the proposed protocol to guarantee termination even in presence of a fully malicious minority of identity servers. The authors of [17] achieve survivable token release by signing authentication tokens through an original RSA-based threshold signature scheme. However, the lack of flexibility of threshold signatures represents a major problem as they force the identity provider to set the value of the security threshold at setup time. Moreover, threshold signatures cause management issues as they cannot guarantee backwards compatibility with non-survivable SSO protocols.

The proposed protocol guarantees flexible SSO and offers the possibility of choosing the value of the security threshold at verification time. This allows a service provider to offer multiple services with different identity assurance levels (e.g., [3, 63]) and to choose the most suitable threshold for each of them. Moreover, it is possible to dynamically adjust the security threshold depending on user contextual information, as suggested by the zero trust paradigm [104], and to tailor the best trade-off between performance and security for each service. Finally, the proposed flexible and survivable SSO can preserve compatibility with non-survivable SSO. This would enable a gradual transition of existing service providers towards survivable SSO. A service provider can enable backwards compatible support to survivable SSO by updating the authentication token verification algorithm.

The password-based survivable SSO proposed in [123] obtains a better trade-off in terms of threshold configuration and survivability with respect to [5] and [17]. It allows the configuration of the security threshold at setup time and guarantees survivability. Although the authors do not explicitly specify an adversarial model, their proposal seems to consider a mobile adversary similar to that proposed in this work. While their protocol obtains good trade-offs, it lacks flexibility due to the adoption of threshold signatures during the token authentication phase.

3.3 System Model

We describe the survivable SSO protocol by referring to Figure 3.1 showing the main entities, data and operation flows. The proposed protocol involves four entities: *user*, *service provider*, *identity provider*, and a set of *identity servers*. The user denotes a person who wants to access services and resources maintained by the service provider. The identity provider denotes an authority that defines and operates a set of identity servers to offer the survivable single sign-on protocol. The protocol involves the following types of data:

- *user credentials*: unique information held by each user presented to identity servers for authentication;
- *credentials databases*: data structures independently maintained by identity servers to verify users credentials;
- *partial tokens*: assertions about users identities authenticated by a single identity server;
- *authentication tokens*: assertions about users identities whose authenticity is guaranteed by a subset of identity servers;
- *token signing keys*: secret cryptographic material held by each identity server to authenticate authentication tokens;
- *identity provider certificate*: public cryptographic material used by the service provider to verify authentication tokens;

- *identity provider signing key*: secret cryptographic material used by the identity provider to authenticate the identity provider certificate.

The proposed protocol consists of five operations.

- *Setup*: the identity provider defines the initial set of identity servers, and releases a certificate that authenticates identity servers public keys. We assume that the certificate is distributed to all actors by using orthogonal public key distribution protocols [28];
- *Register*: the user registers his credentials to all identity servers credentials databases;
- *Sign-on*: the user requests an authentication token to identity servers. This operation is composed by the following steps:
  - *Verify credential*: an identity server verifies user credentials against his credential database;
  - *Release partial*: an identity server releases a partial token to an authenticated user;
  - *Combine*: the user combines the partial tokens collected by a threshold of identity servers in an authentication token;
- *Verify token*: the service provider uses the identity provider certificate to verify the authenticity and validity of the authentication token presented by the user;
- *Refresh*: the identity provider updates identity servers secret cryptographic material.

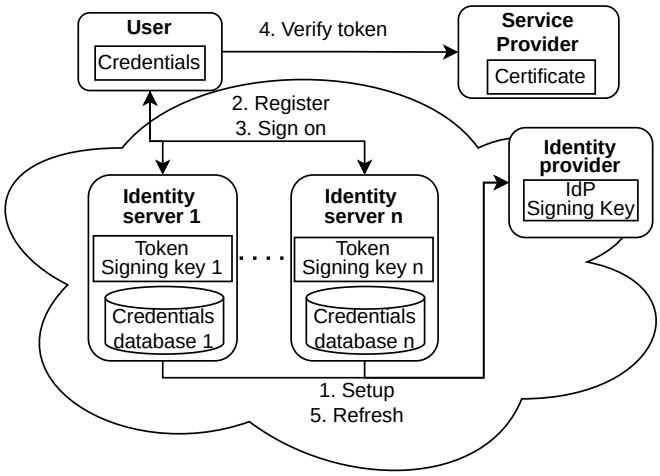

Figure 3.1: Architecture and high level protocol flow

The protocol requires that users establish secure (confidential and authenticated) bidirectional communication channels with legitimate identity servers by using public keys of identity servers distributed during the *Setup* phase. Communication channels allow users to authenticate identity servers, and not vice versa, as it is common for standard HTTPS communications.

The proposed framework represents a novel contribution to model the operations of survivable single sign-on protocols. It captures common operations shared by related proposals which were not highlighted by previous literature, and allows us to compare the proposed protocol with related works [5, 17, 123]. The framework extends existing non-survivable single sign-on frameworks by introducing the Release partial and Combine procedures. Existing related survivable SSO protocols adopt a similar system model where identity servers are coordinated through decentralized protocols which do not require an identity provider [5, 17, 123]. However, they do not guarantee flexibility (see Section 3.6). The proposed architecture introduces the additional role of the identity provider because the considered scenarios, such as cloud-based SSO, are characterized by centralized governance where the identity provider acts as an authority that operates identity servers during Setup and Refresh operations. At Setup time, the identity provider defines the infrastructure while at Refresh time it proactively secures it.

The proposal does not limit the identity provider from being a decentralized entity because it can be extended to support decentralized execution of the Setup and Refresh operations by leveraging ideas from [96]. For ease of presentation and without loss of generality, in the remainder of this chapter we consider the identity provider as one entity.

We enable identity providers to offer flexible and survivable SSO as a service. An identity provider can execute the Setup operation by defining the maximum security threshold k_{max} and by provisioning an infrastructure of $2k_{max} + 1$ failure-independent identity servers. Service providers can choose the most appropriate security threshold value between zero and k_{max} to enforce survivability on their services. A security threshold equal to zero maintains compatibility with existing non-survivable SSO. A security threshold equal to k_{max} allows service providers to enforce the highest level of identity assurance even in presence of k_{max} compromised identity servers. Guaranteeing a practical failure independence of all servers against benign and malicious faults tends to become quickly an intractable challenge as demonstrated in [54, 67]. Hence, we can assume that in practice the value of k_{max} is at most of few units. If we accept stronger security assumptions on identity servers, then the value of k_{max} can be increased above the few units. For example, some identity servers may share the same operating system. This security trade-off simplifies the technological challenge of provisioning and maintaining several operating systems to enable the deployment of a larger yet less diverse set of identity servers.

3.4 Threat model

We discuss possible attacks from a twofold perspective: we discuss violation and recovery patterns throughout the protocol lifetime; we analyze the multiple

classes of attacks that an adversary can adopt to subvert the protocol security. We use these analyses to assess the security guarantees of the proposal in Sections 3.6 and 3.7.

For the threat analysis, we consider the popular *mobile adversary model* that aims at subverting the survivable single sign-on protocol. This model was proposed in the context of distributed function evaluation [99] and applied to secret sharing [65], multiparty computation [47], intrusion tolerant certification authorities [124], key management systems [42], cloud-based secure logging [102] and secure software update systems [91]. It assumes that identity servers are failure-independent and that at any instant an identity server is either *honest* or *compromised*. A compromised identity server can be recovered and become honest after that its hardware and software have been reset to a known clean state and its secret cryptographic material has been obsoleted. A recovery of all identity servers is periodically executed by the identity provider during the Refresh operation. The proposed protocol tolerates that a minority of identity servers is compromised simultaneously, that is, it guarantees security against adversaries that violate up to $k < n/2$ identity servers, where n is the total number of identity servers. This is the typical security level guaranteed by related works using the mobile adversary model (e.g., [65, 42]).

We discuss violation and recovery patterns between periodic Refresh operations by referring to Figure 3.2. The time horizon is divided in *time periods*, where each begins with the execution of the Refresh operation. We denote the remaining part of the time period after the completion of the Refresh as *operation period*. During the operation period, honest identity servers operate the single sign-on protocol according to initial specification. Each highlighted area represents a time interval during which the adversary has compromised up to k identity servers.

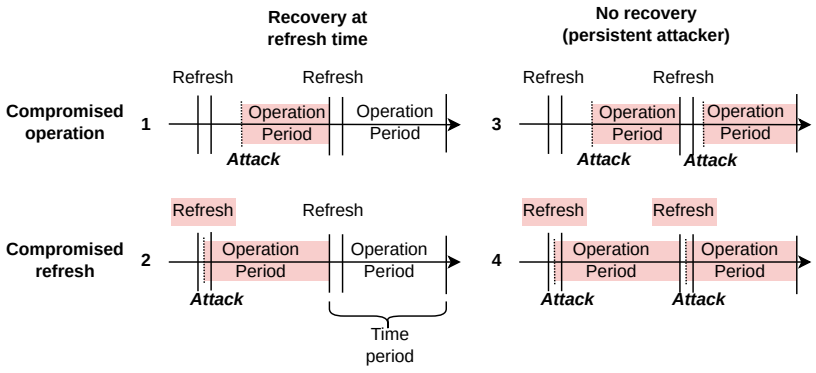

Figure 3.2: Violation patterns

The mobile adversary model considers four attack patterns. The first pattern captures an adversary that has corrupted up to k identity servers during the operation period and that is removed by the identity provider during the

next Refresh operation. The second pattern considers an adversary that has corrupted up to k identity servers during the Refresh operation. The adversary has the additional power to interfere with the identity provider during Refresh. The third and fourth patterns capture a powerful and elusive adversary that is able to move laterally among identity servers and ensures a persistent presence at the identity provider infrastructure even after Refresh operations. An attacker cannot control more than k identity servers during each period.

An adversary can perform different types of attacks to subvert the survivable single sign-on protocol. We label attacks to reference each of them when analyzing security guarantees in Sections 3.6 and 3.7.

- **A.** Violation of the token release system within identity servers:
  - **A1**: the adversary violates the confidentiality of the token signing keys within identity servers to forge authentication tokens;
  - **A2**: the adversary violates the integrity of the token release system by returning bogus partial tokens to users;
  - **A3**: the adversary violates the availability of the token release system by deleting token signing keys or by not returning partial tokens to users.
- **B.** Violation of the identity verification protocols within identity servers:
  - **B1**: the adversary compromises the integrity of the credentials database of an identity server to forcefully set known credentials to users accounts;
  - **B2**: the adversary violates the availability of the identity verification protocol by deleting the credential database or by not completing credential verification;
  - **B3**: the adversary violates the confidentiality of the credentials database of an identity server to recover users credentials;
  - **B4**: the adversary reads an identity server internal state during credential verification to recover users credentials;
  - **B5**: the adversary executes Man-in-The-Middle attacks from compromised identity servers to impersonate the user at honest identity servers during the Sign-on operation.
- **C.** Violation of the identity provider:
  - **C1**: the adversary violates the confidentiality of the identity provider signing key.

3.5 Survivable token release

A comprehensive analysis of the proposed SSO protocol is in Section 3.7. Here, we focus on the token release scheme. The key insight of the proposed token release scheme is twofold. First, signing authentication tokens through conventional digital signatures allows to achieve flexibility. Second, proactively rotating token signing keys allows to guarantee security against lateral movement of the mobile adversary while the identity provider allows to efficiently authenticate rotated public keys and modified system parameters. We consider a black box digital signature scheme defined by the following operations framework:

- $\langle sk, pk \rangle \leftarrow \mathsf{KeyGen}()$: generate secret key sk and public key pk;
- $\sigma \leftarrow \mathsf{Sign}(sk, m)$: compute signature σ on message m with secret key sk;
- $0 \vee 1 \leftarrow \mathsf{Verify}(pk, m, \sigma)$: if signature σ authenticates message m under public key pk output 1, 0 otherwise.

We assume that the identity provider has established the security level of the adopted cryptographic schemes and that the resulting public parameters are known to all actors. For ease of notation, we omit public parameters from the scheme operations. The proposed token release scheme implements the following operations.

$crt \leftarrow \mathsf{Setup}(sk_{IdP}, k_{max}, \{\langle pk_1, \pi_1 \rangle \ldots, \langle pk_n, \pi_n \rangle\})$. The identity provider, given his secret key sk_{IdP} and security threshold k_{max} verifies the identity servers certificate signing requests $\{\pi_1, \ldots, \pi_n\}$ and authenticates the corresponding public keys $\{pk_1, \ldots, pk_n\}$ by computing the identity provider certificate crt. To this aim it executes the following steps:

- the service provider defines the value k_{max} and a set of $n = 2k_{max} + 1$ identity servers;
- each identity server executes the $\mathsf{KeyGen}()$ algorithm to compute the signing key pair $\langle sk_i, pk_i \rangle$;
- each identity server i then computes a certificate signing request π_i for pk_i, and sends π_i to the identity provider over a secure channel. Certificate signing requests can be computed with well-established standard algorithms [98].
- the identity provider collects the public keys $PK = \{pk_1, \ldots, pk_n\}$ and verifies each certificate signing request in $\{\pi_i\}$;
- if all received certificate signing requests $\{\pi_i\}$ are valid, the identity provider authenticates the corresponding set of public keys and the value k_{max} by signing the pair $\langle PK, k_{max} \rangle$ with his private key sk_{IdP}, producing $crt = \langle PK, k_{max}, \sigma_{crt} \rangle$;
- the identity provider publicly distributes crt.

The authentication methods used in procedures Register and VerifyCredential are orthogonal to the proposed token release scheme in terms of functionality, hence we can omit details. A comprehensive discussion about integration with different user authentication methods is in Section 3.7.

$\sigma_i \leftarrow \mathsf{ReleasePartial}(sk_i, id)$: identity server i with secret key sk_i, given identity id produces partial token σ_i. The ReleasePartial procedure assumes that identity id has been already verified during VerifyCredential. The identity server signs id with his secret key sk_i by executing algorithm $\mathsf{Sign}(sk_i, id)$ of the signature scheme. The identity server then sends the resulting signature σ_i to the user over a secure channel.

$\langle id, \{\sigma_i\}, L\rangle \leftarrow \mathsf{Combine}(id, \{\sigma_i\}_{i\in L})$: the user verifies partial tokens $\{\sigma_i\}_{i\in L}$ collected from the set of identity servers L, and outputs the authentication token $\langle id, \{\sigma_i\}, L\rangle$. To this aim he executes the following steps:

- when the user has collected $k+1$ partial tokens, as required by the service provider, the user determines the set of servers L that produced the collected partial tokens;
- the user then verifies that each of the collected partial tokens is authentic, by executing the $\mathsf{Verify}(pk_i, id, \sigma_i)$ for each $i \in L$;
- if all partial tokens are authentic, the user outputs the authentication token $\langle id, \{\sigma_i\}, L\rangle$.

$0 \vee 1 \leftarrow \mathsf{VerifyToken}(\langle id, \{\sigma_i\}, L\rangle, crt, k)$: the service provider verifies whether all signatures in $\{\sigma_i\}$, produced by the set of identity servers L, authenticate id by using crt. It outputs 0 if any σ_i does not authenticate id or if $|L| < k+1$, 1 otherwise. To this aim it executes the following steps:

- the service provider verifies the authenticity of the identity provider certificate $crt = \langle PK, k_{max}, \sigma_{crt}\rangle$ by verifying signature σ_{crt}. The result of this operation can be cached as long as the set PK is not refreshed;
- it verifies that $|L| \geq k+1$;
- it verifies all signatures in $\{\sigma_i\}$ by executing $\mathsf{Verify}(pk_i, id, \sigma_i)$;
- if any of the previous checks fails it outputs 0, otherwise it outputs 1.

$crt' \leftarrow \mathsf{Refresh}(sk_{IdP}, crt, PK_a, PK_r)$: the identity provider sets the new identity provider certificate crt', given the current identity provider certificate crt, the new set of public keys PK_a and the set of revoked public keys PK_r. To this aim he executes the following steps:

- the identity provider recovers all compromised identity servers and establishes the set of additional identity servers, if any.
- each additional or recovered identity server in the new set computes his signing key pair $\langle sk_i, pk_i\rangle$ by executing the $\mathsf{KeyGen}()$ algorithm, produces the corresponding certificate signing request π_i and sends it to the identity provider.

- the identity provider collects the set of new public keys PK_a, verifies the corresponding certificate signing requests $\{\pi_i\}_{i \in |PK_a|}$, and defines the set PK_r of public keys to revoke. PK_r includes the old public keys of recovered servers and any disposed server that is excluded from the protocol.

- the identity provider sets the new set of public keys $PK' = PK \cup PK_a \setminus PK_r$ such that $|PK'| = |PK|$, and authenticates the pair $\langle PK', k_{max} \rangle$ by signing it with his private key, producing $crt' = \langle PK', k_{max}, \sigma_{crt'} \rangle$ which is made publicly available.

3.6 Security guarantees

In this section we discuss the security of the token release scheme against attack classes A and C that we presented in Section 3.4. We devote Section 3.7 to consider class B attacks and evaluate the security of the overall SSO protocol when the token release scheme is integrated with different credential management systems. Here, we show how to mitigate class C attacks that violate the confidentiality of the identity provider signing key. Moreover, we show that the proposed token release scheme is secure against class A attacks that aim to violate the token release system within identity servers. Furthermore, we show that security against class A attacks holds during all violation patterns (Figure 3.2) that is, violation during: one operation period (1), one Refresh execution (2), consecutive operation periods (3) and consecutive Refresh executions (4).

A1. The attacker can violate the confidentiality of the token signing key of up to k identity servers through any of the violation patterns. The attacker can force a compromised identity server to execute the ReleasePartial operation on identities of the attacker's choice. We note that the attacker can obtain the same level of violation even if it only controls the key without knowing its value (e.g. the key is protected by an HSM). Attack A1 with violation pattern 1 is ineffective because an attacker that controls no more than k identity servers is not able to forge a valid authentication token. Attack A1 is ineffective even with the violation pattern 2. Here, the identity provider verifies that the certificate signing request submitted by identity servers is legitimate before certifying the new public keys. Moreover, an adversary that has access to the new secret keys produced during Refresh execution is unable to obtain a valid authentication token since he does not control enough $(k + 1)$ identity servers. Finally, attack A1 is also ineffective with violation patterns 3 and 4. Given that identity servers are recovered at the beginning of the Refresh procedure, the adversary can never control more than k identity servers within any time period. Therefore, the adversary does not control enough $(k + 1)$ identity servers to forge authentication tokens. Moreover, the public keys of the recovered identity servers are revoked during Refresh. We note that the attacker may try to exploit race conditions during revocation of public keys (e.g., CRL propagation delays) to force the service provider to verify authentication tokens with revoked public keys. To prevent this type of attacks we rely on the identity provider as an online certification authority. In this way the service provider always validates authentication tokens with a fresh copy of the new set of public keys PK'.

As a result, the service provider can easily detect invalid authentication tokens authenticated by revoked public keys.

A2. Attack A2 is ineffective in any violation pattern because the proposed token release scheme allows the user to detect bogus partial tokens, discard them and repeat the *Sign-on* operation with another honest identity server; its existence is guaranteed by to the presence of an honest majority.

A3. Attack A3 is ineffective in any violation pattern because the Refresh operation ensures that at any moment there is an available honest majority of $k + 1$ identity servers that is able to respond to users.

C1. The attacker may try to violate the identity provider signing key. However, we note the identity provider secret key sk_{IdP} can be kept offline as it must be used only during Setup and Refresh operations which occur on a much broader frequency than operations involving the identity servers signing keys. As a result, the signing key of identity provider can be protected to ensure it less vulnerable than the signing keys of identity servers. We remind that the survivability of the proposal could be extended to the identity provider by instantiating it as a collective entity as already discussed in Section 3.3.

The proposed protocol guarantees also the *accountability* security property that is, it allows an identity provider to attribute protocol deviations to specific compromised identity servers. This property is important because it allows an identity provider to prioritize recovery operations when servers are compromised, and can be complementary to already deployed approaches for monitoring system infrastructure [10]. All cryptographic material issued by identity servers is accountable. For example, each partial token σ_i produced during the ReleasePartial procedure is accountable because it can be verified through the public key of identity server i. Moreover, the authentication token $\langle id, \{\sigma_i\}, L\rangle$ computed during the Combine procedure is accountable because the set of signers L explicitly indicates the identity servers that contributed to signatures $\{\sigma_i\}$. It is important to note that related works [5, 17, 123] relying on threshold signatures are not completely accountable. Partial tokens computed by identity servers during the *Release partial* procedure may be accountable depending on the adopted scheme, whereas authentication tokens are not accountable. Other papers (e.g., [5, 123]) adopt the threshold signature scheme of Boldyreva [30] which allows accountability of partial tokens because identity servers publish a commitment of their secret shares after the *Setup* and *Refresh* procedures. The work in [17] proposes an original RSA-based threshold signature scheme which blinds partial tokens thus preventing partial token accountability. The authentication token computed during the *Combine* procedure of [5, 17, 123] is not accountable because a key property of threshold signatures is that they do not reveal the identity of individual signers but only the cardinality of the set of signers [30].

The proposed original token release scheme guarantees several flexibility benefits: adjustable security threshold, adjustable performance overhead and compatibility with non-survivable SSO, that we discuss below. The proposed token release scheme guarantees an *adjustable security threshold* because the service provider can decide the value of k during the VerifyToken operation. This allows

the choice of the best trade-off between security and performance for the service he offers, provided that $0 \leq k \leq k_{max} = \lfloor \frac{n}{2} \rfloor$. The constraint $k_{max} = \lfloor \frac{n}{2} \rfloor$ guarantees availability in case of k_{max} disconnected identity servers.

The scheme also guarantees an *adjustable performance overhead* because it allows the service provider to adjust the value of k to the best trade-off between performance and security. Higher values of k imply higher security in terms of token unforgeability and availability, as the adversary is required to violate $k+1$ identity servers to issue rogue authentication tokens or impede their release. However, these higher k values imply higher overheads because a user executes the sign-on procedure with $k+1$ servers. Previous proposals [5, 17, 123], which are based on threshold signatures, do not achieve a comparable flexibility because the value k is not decided by the service provider, but by the identity provider during the *Setup* operation. This value cannot be changed afterwards.

As a final attribute, the proposed token release scheme preserves *compatibility with non-survivable SSO*. If a service provider sets $k = 0$ during the VerifyToken operation, its users execute the SSO procedure with one identity server as in traditional non-survivable SSO.

3.7 Integration with credentials verification and storage protocols

We consider different credentials storage and verification protocols to evaluate their impact on the security of the SSO protocol. To this aim, we consider the attack category B described in Section 3.4, which involve credentials verification and storage protocols as follows:

- B1: integrity violation of credentials database;
- B2: unavailable identification protocol;
- B3: credentials database leak;
- B4: internal state leak;
- B5: Man-In-The-Middle (MITM) attacks from compromised identity servers.

We do not consider impractical credential storage and verification protocols that require users to maintain multiple credentials for identification, even if this is a naive solution to achieve survivable SSO.

We consider the following categories of authentication protocols that can be adopted to build a survivable SSO system:

- **P1**: a strawman approach based on plaintext storage and verification;
- **P2**: approaches that protect password storage but where verification is operated in plaintext [26];
- **P3**: approaches where passwords are protected at verification and storage time [66];

- **P4**: approaches that use secret sharing techniques to distribute passwords over multiple servers [5, 17, 123];

The presence of a majority of honest servers guarantees security against attacks B1 and B2 independently of the adopted authentication protocol and violation pattern. Even if the adversary registers malicious credentials (B1), he is unable to obtain enough valid partial tokens. Moreover, if the adversary prevents the completion of the authentication protocol in compromised identity servers (B2), the remaining honest majority ensures availability. If an authentication protocol is vulnerable to attacks B3 and B4, then the adversary can recover user credentials and impersonate him by compromising one identity server. Hence, violation patterns are not relevant to evaluate the security of protocols that are vulnerable to B3 and B4 because they already fail with one compromised sever.

P1. As a strawman approach, we consider a protocol in which each identity server stores and verifies plaintext passwords. The user sends the password over the secure channel to the identity server which verifies that it matches the stored password. This protocol yields a SSO protocol that is vulnerable to attacks B3, B4 and B5. The protocol is not survivable as an adversary that either compromises a single credentials database (B3) or observes the password verification procedure of one identity server (B4), can naively recover the password. The protocol is vulnerable to R5 (MITM) because, even if the communication channel is authenticated, the adversary may forward messages from compromised identity servers which are legitimate endpoints of the authenticated channel.

P2. Protocols that protect password storage but verify passwords in plaintext rely on password-hashing techniques [26]. Adopting these protocols in the considered distributed SSO scenario requires the user to transmit his password over a secure channel to each identity server which compares the password with its corresponding digest. This protocol is not completely secure against attack B3 because an adversary that captures the credentials database of any identity server can mount offline dictionary attacks (ODA). This may allow the attacker to recover the password to impersonate the user at the other identity servers. Moreover, the resulting SSO protocol is vulnerable to attacks B4 and B5 if an identity server is compromised. This protocol is vulnerable to B4 because a compromised identity server receiving the plaintext password can impersonate the user. It is also vulnerable to B5 because an adversary can forward the received plaintext password and impersonate the user to other identity servers.

P3. Security against server violation can be achieved through authentication protocols where identity servers never access plaintext passwords. The current state-of-the-art is represented by the OPAQUE scheme [66], which allows a user to store a secret key encrypted with his password at the registration phase. During authentication, only a user who knows the correct password can decrypt the secret key to prove his identity. The scheme adopts an oblivious PRF [53] which does not disclose any information about the password nor the secret key to the identity server during the registration and authentication phases. OPAQUE is not vulnerable to B5 because adopts channel bindings to prevent MITM

	B3 verification key	**B4** internal state	**B5** MITM
P1 - Strawman	○	○	○
P2 - Pwd Hashing	◑	○	○
P3 - OPAQUE	◑	◑	●
P4 - Secret sharing	●	●	●

○: vulnerable, ◑: partially vulnerable (offline dictionary attack), ●: not vulnerable

Table 3.1: Security guarantees of survivable single sign-on systems with different authentication protocols

attacks. While the SSO protocol obtained by different OPAQUE instances with each identity server is not vulnerable to B5, it is not completely secure against B3 and B4 because an adversary that captures the credentials database or observes the verification procedure of a single identity server may be able to mount offline dictionary attacks.

P4. We conclude by evaluating proposals that are secure against attacks B3, B4 and B5 relying on schemes based on secret sharing, such as Threshold Oblivious PRFs (TOPRF) [5, 17, 123]. The work of [5] is secure against these attacks considering static corruptions of identity servers, whereas [17, 123] are secure in all violation patterns as they are proven secure against mobile adversaries. They are secure against B3 and B4 because they rely on techniques based on secret sharing to store and transmit the password. Hence, individual messages and credential databases do not contain enough information to mount attacks that can recover the password, such as offline dictionary attacks. These proposals are also secure against B5 because they are proven secure against active adversaries that can eavesdrop communications of up to a threshold of other identity servers.

The trade-offs of different authentication protocols are summarized in Table 3.1, where columns denote SSO requirements and rows denote the considered protocols. Password-based protocols that are not designed to be survivable, are either insecure against B3, or only partially secure. We note that when a protocol is partially secure against B3 due to possible offline dictionary attacks, the user can choose a strong password to make these attacks ineffective.

3.8 Final remarks

We propose the first flexible and survivable SSO protocol that relies on a distributed architecture of identity servers that collectively authenticate users and issue SSO tokens through a novel scheme. Flexibility allows service providers to choose the best trade-off between performance and security for each service and to preserve compatibility with non-survivable SSO. Survivability allows the identity provider to guarantee a high level of identity assurance even in presence of successful intrusions. We evaluate the security of the overall survivable SSO by considering several state of the art authentication protocols. Moreover, we

show that the proposed token release scheme is secure against a comprehensive set of attack classes. Flexibility and survivability make the proposal a viable solution to offer secure and robust authentication to cloud services and any mission critical system that must rely on SSO, as in the case of survivable Zero Trust Architectures described in Chapter 2. The results of this work are open to different developments. It should be interesting to investigate how emerging passwordless authentication protocols may impact flexibility in the context of survivable SSO systems. Moreover, we think that it is possible to extend this proposal to support decentralized management systems as in the context of multi-cloud environments.

Chapter 4

SPOC: Survivable Passwordless Single Sign-On

4.1 Introduction

Single Sign-On (SSO) is a popular delegated authentication protocol where an identity provider authenticates users that need to prove their identity to service providers. To this aim the identity provider adopts a logical identity server that authenticates users and that reports the authentication outcome by issuing an identity attestation. Recent serious incidents (e.g., [39, 4]) show the vulnerability of this centralized authentication design. Attackers that are able to compromise the identity server can access user credentials or, even worse, issue arbitrary attestations to impersonate users. A recent line of research proposes *survivable* SSO to tolerate intrusions in the identity provider infrastructure [5, 17, 123]. In survivable SSO, the identity provider adopts multiple logical identity servers that collectively issue identity attestations. In such a way, a user must demonstrate identity to a service provider by presenting attestations issued by multiple identity servers. The number of identity servers issuing an attestation must be greater than the number of malicious identity servers that the protocol can tolerate.

To date, only password-based survivable SSO protocols have been proposed [5, 17, 123]. These protocols address the challenge of protecting weak credentials against offline dictionary attacks and identity attestation forgery in presence of intrusions. However, they inherit typical vulnerabilities related to password usage, including phishing attacks targeting user passwords. Passwordless authentication is the last line of defense against a successful phishing attack, yet no existing survivable SSO protocol provides the phishing resistance guarantees offered by passwordless authentication.

We propose SPOC, the first survivable passwordless SSO protocol, that extends the state-of-the-art FIDO2 passwordless authentication protocol and OpenID Connect (OIDC) single sign-on protocol. In our proposal, the identity provider deploys multiple failure-independent identity servers and manages their cryptographic material. A user authenticates at multiple identity servers by using a FIDO2 authenticator through a user agent which collects an identity attestation from each server. The user agent then sends the collected attesta-

tions to the service provider, which accepts the user identity only if the number of valid attestations exceeds a given threshold. The proposed protocol achieves the following benefits. *Security*: it considers the same threat model of FIDO2 and OIDC standards and adds additional security guarantees related to survivability, without weakening any security assumption considered by FIDO2 and OIDC. *Usability*: its distributed nature is transparent to users, and does not hinder usability with regard to the original non-survivable protocols. *Compatibility*: it is compatible with unmodified FIDO2 authenticators; hence, it allows the use of existing hardware security tokens available on the market. *Performance*: it achieves response times that are accepted by established performance metrics for usable authentication systems [62].

As no prior work exists on survivable passwordless authentication, we first identify the novel attack classes that emerge in this context. Then, we design the proposed protocol through a modular approach. We give a formal definition of a *Survivable Passwordless Challenge-response (SPC) protocol* which considers the concurrent execution of multiple challenge-response protocols between the client and each identity server. We propose a specification based on the FIDO2-WebAuthn protocol (*Survivable WebAuthn* (SWA)). Then, we formally define a *Survivable Passwordless SSO (SPS) protocol* by extending the SPC protocol with SSO. We denote as SPOC a specification of SPS based on the combination of SWA and an extension of the implicit flow of OpenID Connect. To prove both SWA and SPOC secure, we formalize security properties that capture both novel attack classes, and existing attack classes that have never been formalized by the literature. This formalization can be of independent interest. Finally, we evaluate usability and performance on a software prototype[1].

Section 4.2 discusses related work. Section 4.3 presents the system model. Section 4.4 discusses overall design challenges. Section 4.5 introduces the SPC protocol and the SWA specification. Section 4.6 describes the SPS protocol and the SPOC specification. Section 4.7 discusses experimental results. Section 4.8 includes examples of novel attacks. Sections 4.9 and 4.10 include SWA and SPOC security proofs. Section 4.11 proofs security of the composition of SPOC with the FIDO2-CTAP2 standard. Section 4.12 concludes the chapter with final remarks on future work.

4.2 Related work

We propose SPOC, the first survivable passwordless authentication protocol that is compatible with the state-of-the art FIDO2 standard for passwordless authentication [1]. As no prior work exists in this area, we are the first to consider passwordless usability requirements in survivable authentication. Recent research shows that non-survivable passwordless authentication such as FIDO2 is considered usable by end-users [85, 49, 76]. As a result, we argue that survivable passwordless authentication protocols should preserve the same usability level of their non-survivable counterparts. In particular, survivable passwordless authentication protocols should require a single user mediation [117] to confirm

[1]The software will be open-sourced in case of acceptance

the user's willingness to complete an authentication procedure, even if authentication involves the execution of a distributed protocol with multiple servers. This requirement rules out strawman implementations that sequentially execute non-survivable authentication protocols with multiple servers, as they would require a single user mediation for each server. We consider security and usability as the most important requirements and we design SPOC accordingly.

We also consider the deployability of SPOC, in the sense of Bonneau et al. framework for evaluating Web authentication schemes [32]. The framework does not consider a novel deployability benefit of *token compatibility* that is relevant to our effort, as we preserve compatibility with existing authenticator implementations. Existing authenticators require no modifications to participate in the SPOC protocol. Moreover, the protocol is also *browser-compatible* in the sense of Bonneau et al. framework, as client-side computation can be executed with standard browsers technologies (e.g., JavaScript) thus not requiring any modification to existing browser software.

This work is closely related to a recent line of research that proposes intrusion-tolerant SSO protocols based on password authentication [5, 17, 123]. PASTA [5] considers an adversary that can corrupt up to a threshold of identity servers during the whole system lifetime. However, this protocol does not define the due procedures to recover compromised identity servers to a safe state. The ability to recover after a successful intrusion is essential in proactively secure and survivable systems [37, 48]. SPOC allows us to recover compromised identity servers so to provide proactive security guarantees. As a result, it tolerates also *mobile* adversaries [99, 121] that perpetually try to compromise a threshold of identity servers in a given time unit and ensure persistent presence by moving laterally among identity servers. Even PESTO [17] considers a mobile adversary under an adaptive corruption model, which gives stronger security guarantees than our static corruption model. However, PESTO authentication does not complete if one identity server is unavailable. On the other hand, through sufficient redundancy SPOC can guarantee completion by tolerating unavailability of a threshold of identity servers. PROTECT [123] also obtains good trade-offs in terms of proactive security and completion guarantees. Although the authors do not formally specify their adversarial model, their proposal seems to consider a mobile adversary. We observe that SPOC requires a trusted third party (the identity provider) to execute system setup and to periodically refresh cryptographic keys by communicating with each identity server. Nonetheless, we remark that these are offline operations, which are typically much harder to compromise, and that SPOC does not have any online attack surface which represents a single point of failure. This design choice allows more efficient protocols and a simpler deployment, and we consider it as an acceptable trade-off for an intrusion-tolerant system with centralized governance. We leave investigating decentralized key refresh for SPOC as a future work. In comparison, PROTECT has a decentralized setup and decentralized key refresh that requires synchronous communications among servers, PESTO has centralized setup and decentralized key refresh which requires no communications among servers, and PASTA has trusted centralized setup (and does not support key refresh). Al-

though all these proposals do not explicitly state assumptions for the considered communication model, they seem to consider the same synchronous and reliable point-to-point communication channels that we consider for SPOC.

The formal model used in this work to prove the security of the proposal is related to the analyses in [16] on FIDO2 [1]. That work adopts the Bellare-Rogway model [20] to formally analyze the security of the WebAuthn [116] and CTAP2 [7] protocols and their composition in the FIDO2 standard. We follow an analogous approach: we adopt the formal model of [16] and use it to extend the security definitions of the original WebAuthn protocol to our distributed scenario. Moreover, we rely on the same security assumptions of existence of collision-resistant hash functions [43] and existentially unforgeable under chosen message attacks signature schemes [61].

For our security definitions we use a related work [51] that formally defines the security properties of the OIDC protocol [107], although their model is incompatible with the model used in this proposal. We translate the formal security properties of OIDC introduced by [51] to our model and extend them to capture distributed SSO security guarantees. In particular, to the best of our knowledge we are the first to consider SSO as a key transport protocol [33] and to formalize its security by considering the Bellare-Rogway requirements for key establishment protocols. The translation to the Bellare-Rogway model and extension to the distributed setting of OIDC security properties may be considered original contributions of independent interest.

4.3 System model and notation

The protocols involve a set of parties $\mathcal{P}$. The set of parties is composed of the following finite, disjoint, non-empty sets: users $\mathcal{U}$, authenticators $\mathcal{T}$, clients $\mathcal{C}$, identity servers $\mathcal{S}$, identity providers $\mathcal{I}$, service providers $\mathcal{V}$. A party $P \in \mathcal{P}$ is named by a string of finite length id which uniquely identifies P in $\mathcal{P}$. We call $F : \mathcal{S} \to \mathcal{I}$ a public surjective function that maps identity servers to identity providers. Function F models a membership relation of an identity server to the administrative domain of an identity provider. Given identifier id_S of identity server $S \in \mathcal{S}$, $F(\mathsf{id}_S)$ returns the identifier id_I of its corresponding identity provider $I \in \mathcal{I}$. We say that identity provider I *controls* or *owns* identity server S if $F(\mathsf{id}_S) \stackrel{?}{=} \mathsf{id}_I$ and we write $\mathsf{id}_S \in \mathsf{id}_I$. We denote the set of all identity servers owned by identity provider I as $\mathcal{S}_I$ and its cardinality as n_I. In practice, identifiers for identity servers and identity providers can be implemented via domain names, and function F returns the domain of a subdomain [94].

Authenticator T maintains an *authenticator key pair* $\langle ak_T, vk_T \rangle$ and a set of *authenticator registration contexts* $\{\mathsf{rct}_T\}$. An authenticator registration context rct_T includes an *authenticator user identifier auid* and a public key credential that is uniquely bound to an identity provider I. Each identity server S stores a set of *server registration contexts* $\{\mathsf{rcs}_S\}$. Each server registration context rcs_S includes the public key and authenticator user identifier of an authenticator registration context rct_T associated with identity provider I that owns server S. Both authenticator and server registration contexts also include other stateful

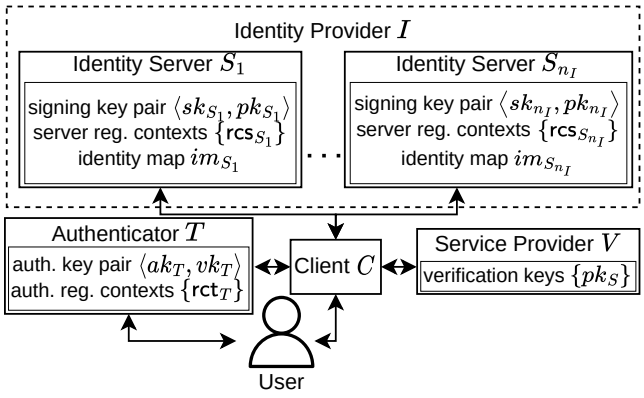

Figure 4.1: System model

information used for authentication procedures. Each identity server also stores an *identity map*, which is an associative array im that maps an authenticator user identifier *auid* to a *user identity uid*, and stores a *signing key pair* $\langle sk_S, pk_S \rangle$ to authenticate identity attestations of user identities. The service provider V maintains a set of *verification keys* $\{pk_S\}_{S \in \mathcal{S}_I}$ for each identity provider I to verify authenticity of identity attestations released by servers $\mathcal{S}_I$ owned by I. Clients do not store persistent information.

We assume a synchronous system where all parties can access a shared clock and adopt synchronous and reliable point-to-point communication channels. Identity provider I divides time into time periods. Each time period is uniquely identified by label ω_I^ℓ in the ordered set of time periods Ω_I, where $\ell \in \mathbb{N}$ denotes the ℓ-th time period. In a time period, each identity server is either malicious or honest. We denote as k_I the number of allowed malicious identity servers during the same time period.

Further notation. We denote as $\{\mathsf{rct}_T\}$ ($\{\mathsf{rcs}_S\}$) the set of all the authenticator (server) registration contexts maintained by authenticator T (server S). We denote as $\langle \mathsf{rcs}_S \rangle_{S \in \mathcal{S}_I}$ the tuple of all the server registration contexts associated with the same authenticator and maintained by all servers in $\mathcal{S}_I$. We denote identities with distinct superscripts (e.g., $\hat{\mathsf{id}}$, $\bar{\mathsf{id}}$, $\tilde{\mathsf{id}}$) and call them *intended identities* to capture attacks that exploit inconsistent views on the identities of protocol participants (e.g., phishing). We use $\leftarrow$ and $\leftarrow\!\!\$$ to assign the output of deterministic and randomized algorithms, and $\perp$ is a special return value that denotes failure. We use $\{a|b\}$ to express mutually exclusive return values a or b. We denote as $\leftarrow\!\!\$ \ \{0,1\}^\lambda$ uniform sampling of bit strings of length λ, where λ denotes the security level parameter (e.g, 128-bit). We denote as $[n]$ the set of integers $\{1, \ldots, n\}$, as Q the set of identity servers that participate in a protocol run, as $|Q|$ its cardinality, as $|Q|_{min}$ the minimum cardinality value required by the protocol, as H a collision-resistant hash function [43], as $\mathsf{Sig} = \langle \mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify} \rangle$ a digital signature scheme [61].

We recall definitions of H and Sig. Let H be a function family $\mathsf{H} = \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$

such that $|\mathcal{D}| > |\mathcal{R}|$ and $\mathsf{H}_k = \mathsf{H}(k, \cdot)$ can be computed efficiently on any input given key $k \in \mathcal{K}$. In the security experiment, adversary $\mathcal{A}$ takes as input a random $k \leftarrow\!\!\$\ \mathcal{K}$ and outputs a pair of messages $\langle a, b\rangle \in \mathcal{D} \times \mathcal{D}$. The adversary advantage $\mathsf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{A})$ against collision-resistance of function H is the probability that $a \neq b \wedge \mathsf{H}_k(a) = \mathsf{H}_k(b)$. Hash function H is *collision resistant* if $\mathsf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{A})$ is negligible. In practice H is not a keyed function family, but consists of a single hash function for which it should be infeasible to construct an efficient adversary against collision resistance.

A signature scheme Sig is a tuple of efficient algorithms $\langle \mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify}\rangle$ defined as follows:

- $\langle sk, pk\rangle \leftarrow \mathsf{Gen}()$: outputs pair $\langle sk, pk\rangle$, where sk is a private signing key, and pk is a public verification key.
- $\sigma \leftarrow \mathsf{Sign}(sk, m)$: outputs signature σ on message m given secret signing key sk.
- $\{0|1\} \leftarrow \mathsf{Verify}(pk, m, \sigma)$: outputs 1 if signature σ for message m is verified by public verification key pk, 0 otherwise.

Correctness requires that for any $\langle sk, pk\rangle \leftarrow \mathsf{Gen}()$ and any m, $\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1$. For security, consider a security experiment between a challenger, and adversary $\mathcal{A}$ that has access to signing oracle $\mathsf{Sign}_{sk}(\cdot) = \mathsf{Sign}(sk, \cdot)$. First, the challenger runs $\langle sk, pk\rangle \leftarrow \mathsf{Gen}()$ and gives pk to $\mathcal{A}$. Then $\mathcal{A}$ outputs a message-signature pair $\langle m, \sigma\rangle$. The adversary advantage $\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathsf{Sig}}(\mathcal{A})$ against existential unforgeability under chosen message attack of signature scheme Sig is the probability that $\mathsf{Verify}(pk, m, \sigma) = 1$ and $\mathcal{A}$ has not queried oracle $\mathsf{Sign}_{sk}(m)$. Signature scheme Sig is $\mathsf{euf-cma}$ secure if $\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathsf{Sig}}(\mathcal{A})$ is negligible.

4.4 Overall design and challenges

As SPOC is a specification of the Survivable Passwordless SSO (SPS) protocol, we overview the design of SPS to also give an intuition of SPOC. We design SPS through a modular approach. First, we design a Survivable Passwordless Challenge-response (SPC) protocol that performs a distributed proof-of-possession among a FIDO2-compliant authenticator and a set of identity servers. Then, we design SPS as an extension of SPC to support SSO architectures. Intuitively, the design of SPC differs from a typical challenge-response protocol because the prover (the pair client/authenticator) proves possession of a secret by authenticating a set of challenges, and the verifier (each identity server) checks that its own challenge belongs to the authenticated set. With regard to a typical SSO protocol, SPS requires the service provider to check that the number of verifiers that accepted an SPC execution exceeds a threshold that depends on the number of malicious identity servers. In the following, we describe the overall operations flow of SPS, we introduce the novel attack classes, and we overview the security experiments structure and novelties.

Overall flow. SPS consists of three protocols: *registration*, *authentication* and *refresh*. Registration allows a set of identity servers to associate an authenticator to a known user identity. Authentication allows users to demonstrate

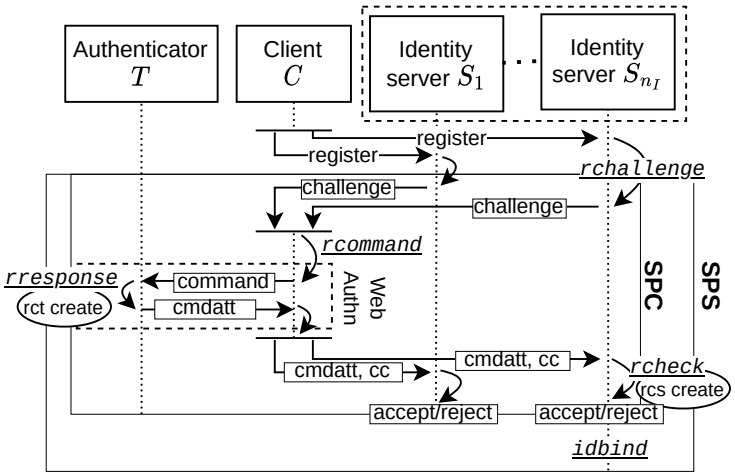

Figure 4.2: SPS registration flow

their identity at a service provider by proving possession of their credentials via the authenticator to a set of identity servers. Refresh allows an identity provider to proactively rotate compromised signing keys and corrupted registration contexts on identity servers. Below we overview registration and authentication to ease understanding of the protocol and of the main contributions.

Figure 4.2 shows SPS registration, which inherits the same four routines of SPC registration and introduces an additional routine. Each identity server executes rchallenge to compute a challenge. The client receives servers challenges and executes rcommand to aggregate them into a command data structure (*command*), which includes the aggregated challenges (*cc*). The authenticator signs the aggregated challenges with rresponse to generate a command attestation (*cmdatt*) and creates a new authenticator registration context (*rct*). The client forwards the command attestation and the aggregated challenges to all identity servers. Each server validates the aggregated challenges with rcheck, creates a new server registration context (*rcs*) and binds the created context to the user identity with idbind.

Figure 4.3 shows SPS authentication, which inherits three routines from SPC and introduces four additional routines. The service provider establishes collective session information by executing abegin, which is separated by the client into distinct session information data, each forwarded to a different identity server. Each identity server generates a challenge with achallenge. The client receives server challenges and executes acommand to aggregate them into a command data structure, which includes the aggregated challenges (*cc*). The authenticator signs the aggregated challenges with aresponse to generate a command attestation (*cmdatt*) and updates the appropriate authenticator registration context. The client forwards the command attestation and the aggregated challenges to the identity servers. Each server runs acheck to verify them and

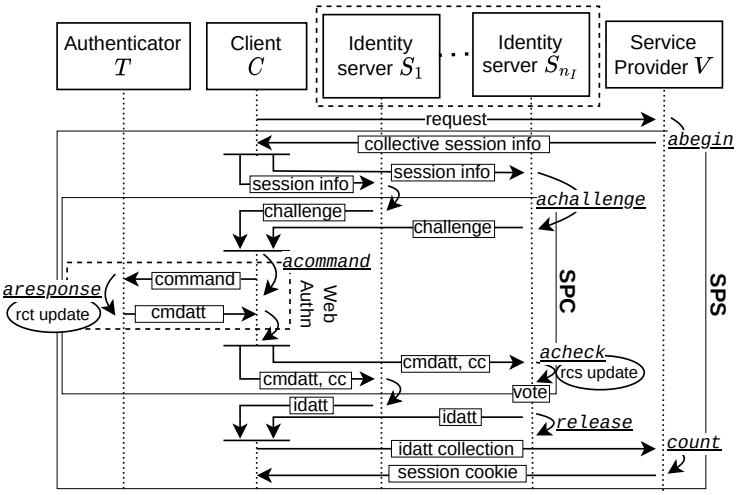

Figure 4.3: SPS authentication flow

updates the registration context. Moreover, each server executes release to issue an identity attestation (*idatt*). The client aggregates identity attestations into a collection and sends it to the service provider. The service provider executes count to verify the identity attestation collection and to issue a session cookie to the client.

We observe that both Registration and Authentication adopt unmodified FIDO2-WebAuthn routines and associated data structures to interact with FIDO2 authenticators (command and response routines within dashed boxes in Figures 4.2 and 4.3). The other routines and data structures have been modified to support the decentralized setting (see Sections 4.5.1 and 4.6.1).

Novel security threats. To define the security of SPS we extend existing *authentication* and *session integrity* security properties of *non-survivable* SSO protocols [51] by considering a *mobile* adversary [99, 121] that can compromise a subset of identity servers within a time period. The *authentication* property refers to the inability of an adversary to impersonate a user at a service provider without compromising the identity provider (e.g., obtaining an identity attestation via a compromised communication channel). The *session integrity* property refers to the inability of an adversary to authenticate a user: i) at a service provider without the user's explicit consent (e.g., a replay attack), or ii) under an identity that is different from the identity proved to the identity provider (e.g., cross-site request forgery attacks).

We identify three novel security threats: *clone detection evasion*, *attestation mixing* and *shared session* attacks. We describe their impact on *authentication* and *session integrity* and hint at how SPS defends against them. Section 4.8 includes visual examples of these threats. In Section 4.5.2 and 4.6.2 we design novel security games to capture these attacks and design SPS accordingly.

Authenticator clone detection evasion consists in the undetectable creation

of multiple instances of the same credentials to perform stealth impersonation. In FIDO2, the identity server guarantees authenticator clone detection by relying on signature counters stored in registration contexts. We show that clone detection in a distributed environment is a harder challenge which cannot be exclusively enforced by identity servers alone. Since a subset of identity servers may be malicious, adversarial interleavings of authentication sessions may evade clone detection. SPS defends against clone detection evasion by: i) cryptographically binding each identity attestation to a SPC session; ii) requiring the service provider to verify that the number of legitimate identity attestations is at least $2k_I + 1$, and that they are bound to the same SPC session; iii) limiting the total number of identity servers to $[2k_I + 1, 3k_I + 1]$. We note that while the $2k_I + 1$ lower bound is required to guarantee the overall security, the $3k_I + 1$ upper bound is specific to prevent *clone detection evasion*.

An *attestation mixing* attack refers to the ability of an adversary to mix and replay identity attestations collected from multiple sessions in different time periods, possibly issued by malicious identity servers, to exceed the allowed security threshold of malicious servers and break *authentication*. SPS defends against this attack by scoping each attestation to a time period through a cryptographic binding.

A *shared session* attack refers to the ability of an adversary to inject an identity attestation collection in the victim authentication session to authenticate the victim under the adversary's identity, thus breaking *session integrity*. This attack is possible if the service provider uses the same session information for all identity servers. The adversary can obtain the victim's session information via a malicious identity server, and start a concurrent SPC authentication session to obtain valid identity attestations of the adversary identity which however are bound to the victim's session. The victim session can then be forced to use the adversary attestations via injection attacks. SPS defends against shared session attacks by requiring the service provider to generate distinct and unpredictable session information for each identity server.

Security experiments. We define SPS security with three security experiments. Experiment 1 in Section 4.5.2 and Experiment 3 in Section 4.6 capture security of SPC and SPS respectively. Experiment 3 extends Experiment 1 as we show that SPC security is a necessary condition to SPS security. To prove that the proposed SPC specification (SWA) is a secure SPC protocol in the sense of Experiment 1, we also define Experiment 2 in Section 4.5.3 to capture attacks against the SWA specification of the SPC Refresh protocol. The experiments capture both the novel attack classes outlined above as well as known threats related to authenticator cloning and secure session cookie establishment that were not formalized in previous literature. To capture authenticator cloning in Experiments 1 and 3 we introduce a Clone query. To capture secure session cookie establishment in Experiment 3 we adopt Reveal and Test queries introduced by Bellare and Rogaway [20] in the context of secure key establishment.

4.5 Survivable Passwordless Challange-response (SPC) protocol

4.5.1 SPC operations framework

The proposed Survivable Passwordless Challenge-response (SPC) protocol is composed of the *key generation*, *register*, *authenticate* and *refresh* subprotocols.

Key generation: $\langle ak_T, vk_T \rangle \leftarrow\!\$\ \mathsf{Spc.Kgen}()$: executed once for each authenticator T to generate the authenticator key pair $\langle ak_T, vk_T \rangle$.

Register (Spc.Register): allows a client to register an authenticator T (initialized with authenticator key pair $\langle ak_T, vk_T \rangle$) at identity provider $\bar{\mathsf{id}}_I$, where $\bar{\mathsf{id}}_I$ denotes the identity provider intended identity observed by the client (see Section 4.3). The client interacts with a set of identity servers $\mathcal{S}_I$ to establish the authenticator and server registration contexts rct_T and $\langle \mathsf{rcs}_S \rangle_{S \in \mathcal{S}_I}$. Register is composed of the following routines:

- $rc_S \leftarrow\!\$\ \mathsf{Spc.rchallenge}(\mathsf{id}_S)$: executed by each identity server $S \in \mathcal{S}_I$, it generates a challenge rc_S, given the server identity id_S.
- $\langle M_r, cc, auid \rangle \leftarrow \mathsf{Spc.rcommand}(\bar{\mathsf{id}}_I, \langle rc_S \rangle_{S \in \mathcal{S}_I})$: executed by the client, it generates authenticator user id $auid$, collective challenge cc and registration command M_r, given challenges $\langle rc_S \rangle_{S \in \mathcal{S}_I}$ and intended identity $\bar{\mathsf{id}}_I$.
- $\langle R_r, \{\mathsf{rct}_T\}' \rangle \leftarrow\!\$\ \mathsf{Spc.rresponse}(ak_T, \{\mathsf{rct}_T\}, M_r)$: executed by authenticator T, returns registration response R_r and the updated authenticator registration contexts $\{\mathsf{rct}_T\}'$, given authenticator private key ak_T, the existing authenticator registration contexts $\{\mathsf{rct}_T\}$ and registration command M_r.
- $\{\mathsf{rcs}_S\}' \leftarrow \mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_T, rc_S, R_r, cc, auid)$: executed by each identity server S, returns the updated server registration contexts $\{\mathsf{rcs}_S\}'$, given provider and server identities id_I and id_S, the existing server registration contexts $\{\mathsf{rcs}_S\}$, authenticator public key vk_T, server challenge rc_S, registration response R_r, collective challenge cc and authenticator user identifier $auid$.

Authenticate (Spc.Authenticate): allows a client to authenticate to a subset of identity servers $Q \subseteq \mathcal{S}_I$ of the identity provider $\hat{\mathsf{id}}_I$ by using registered authenticator T, which update the previously established server and authenticator registration contexts $\langle \mathsf{rcs}_S \rangle_{S \in Q}$ and rct_T. We assume that the client can obtain identities (i.e. FQDNs) of servers owned by provider $\hat{\mathsf{id}}_I$, which the client uses to contact them and start execution of the protocol (e.g., the service provider of SPS returns the list of identities in the abegin routine, see Section 4.6.1). Authenticate is composed of the following routines:

- $ac_S \leftarrow\!\$\ \mathsf{Spc.achallenge}(\mathsf{id}_S)$: executed by each identity server S, returns challenge ac_S given identity id_S.

- $\langle M_a, cc\rangle \leftarrow \mathsf{Spc.acommand}(\hat{\mathsf{id}}_I, \langle ac_S\rangle_{S\in Q})$: executed by the client, it generates collective challenge cc and authentication command M_a, given challenges $\langle ac_S\rangle_{S\in Q}$ and provider identity $\hat{\mathsf{id}}_I$.
- $\langle R_a, \{\mathsf{rct}_T\}'\rangle \leftarrow\!\!\$\ \mathsf{Spc.aresponse}(\{\mathsf{rct}_T\}, M_a)$: executed by authenticator T, returns authentication response R_a and the updated authenticator registration contexts $\{\mathsf{rct}_T\}'$, given registration contexts $\{\mathsf{rct}_T\}$ and authentication command M_a.
- $\langle b_S, \{\mathsf{rcs}_S\}'\rangle \leftarrow \mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R_a, cc)$: executed by each identity server S, returns vote b_S (accept if server S accepts, reject otherwise) and the updated server registration contexts $\{\mathsf{rcs}_S\}'$, given provider and server identities id_I and id_S, registration contexts $\{\mathsf{rcs}_S\}$, challenge ac_S, authentication response R_a and collective challenge cc.

Refresh: $\{\mathsf{rcs}\}' \leftarrow \mathsf{Spc.Refresh}(\mathsf{id}_I, k_I, \{\mathsf{rcs}_{S_1}\}, \ldots, \{\mathsf{rcs}_{S_{n_I}}\})$: executed at the beginning of each time period $\omega_I^\ell \in \Omega_I$ by identity provider identified by id_I on all of its identity servers $\mathcal{S}_I$, it terminates pending registration and authentication sessions. It returns a new set of server registration contexts $\{\mathsf{rcs}\}'$, given security threshold k_I and server registration contexts $\{\mathsf{rcs}_S\}$ of all identity servers $S \in \mathcal{S}_I$.

Intuitively, SPC correctness requires that each server $\mathsf{id}_S \in Q \subseteq \mathcal{S}_I$, even after multiple $\mathsf{Spc.refresh}$ executions, always accepts an authentication that is consistent with a prior registration ($\hat{\mathsf{id}}_I = \bar{\mathsf{id}}_I$) if and only if the identity server belongs to the client's intended identity provider ($\mathsf{id}_S \in \hat{\mathsf{id}}_I$).

Correctness (SPC). Correctness requires that for any identity provider identities id_I, $\hat{\mathsf{id}}_I$, $\bar{\mathsf{id}}_I$ and server identity id_S, for all $k_I, n_I \in \mathbb{N}$ such that $k_I < n_I$ and for all $S \in Q \subseteq \mathcal{S}_I$ if:

$$
\begin{array}{l}
\langle ak_T, vk_T\rangle \leftarrow\!\!\$\ \mathsf{Spc.Kgen}() \\
rc_{S_i} \leftarrow\!\!\$\ \mathsf{Spc.rchallenge}(\mathsf{id}_{S_i}), \forall i \in [n_I] \\
\langle M_r, cc, auid\rangle \leftarrow \mathsf{Spc.rcommand}(\bar{\mathsf{id}}_I, \langle rc_{S_i}\rangle_{i\in[n_I]}) \\
\left\langle R_r, \{\mathsf{rct}_T\}'\right\rangle \leftarrow\!\!\$\ \mathsf{Spc.rresponse}(ak_T, \{\mathsf{rct}_T\}, M_r) \\
\{\mathsf{rcs}_{S_i}\}' \leftarrow \mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_{S_i}, \{\mathsf{rcs}_{S_i}\}, vk_T, rc_{S_i}, R_r, cc, auid), \forall i \in [n_I] \\
\{\mathsf{rcs}\}' \leftarrow \mathsf{Spc.Refresh}(\mathsf{id}_I, k_I, \{\mathsf{rcs}_{S_1}\}, \ldots, \{\mathsf{rcs}_{S_{n_I}}\}) \\
ac_{S_i} \leftarrow\!\!\$\ \mathsf{Spc.achallenge}(\mathsf{id}_{S_i}), \forall i \in [|Q|] \\
\langle M_a, cc\rangle \leftarrow \mathsf{Spc.acommand}(\hat{\mathsf{id}}_I, \langle ac_{S_i}\rangle_{i\in[|Q|]}) \\
\left\langle R_a, \{\mathsf{rct}_T\}'\right\rangle \leftarrow\!\!\$\ \mathsf{Spc.aresponse}(\{\mathsf{rct}_T\}, M_a) \\
\left\langle b_{S_i}, \{\mathsf{rcs}_{S_i}\}'\right\rangle \leftarrow \mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_{S_i}\}, ac_{S_i}, R_a, cc), \forall i \in [|Q|]
\end{array}
$$

then the condition:

$$b_{S_i} = (\mathsf{id}_I \stackrel{?}{=} \bar{\mathsf{id}}_I) \wedge (\mathsf{id}_I \stackrel{?}{=} \hat{\mathsf{id}}_I) \wedge (\mathsf{id}_{S_i} \in \hat{\mathsf{id}}_I), \forall i \in [|Q|] \tag{4.1}$$

holds with probability 1.

We highlight that condition (4.1) prevents an honest authenticator to be partnered with identity servers belonging to distinct identity providers.

Comparison with WebAuthn. We observe that the Spc.rresponse and Spc.aresponse routines are compliant with FIDO2-WebAuthn, making the framework compliant with FIDO2 authenticators interfaces. The two routines output only a single response, thus registration response R_r is the same for all identity servers in $\mathcal{S}_I$, and authentication response R_a is the same for all identity servers in $Q \subseteq \mathcal{S}_I$. Instead, we modify the other interfaces: while, in FIDO2-WebAuthn the *auid* value is generated by rchallenge, in the SPC framework it is generated by Spc.rcommand. This modification avoids distributed generation of *auid* among multiple identity servers, which would require consensus. Adversarial *auid* values are not a problem: duplicate *auid* values are rejected by honest servers, and new inconsistent *auid* values do not affect existing registration contexts and only render completion of future authentications impossible to byzantine client-authenticator pairs.

4.5.2 SPC security model

Trust assumptions. We assume that communication channels between authenticators and clients, and between clients and identity servers are not authenticated, nor private. Authenticators are tamper-proof: the adversary can only read the authenticator internal state and cannot write it.

Session oracles. As in [16], during execution there may be many instances of party $P \in \mathcal{T} \cup \mathcal{S}$. We call instance i of party P a *session oracle*, and we denote it as $\pi_P^{i,j}$. When $j = 0$, $\pi_P^{i,0}$ is party P i-th registration session. When $j \geq 1$, $\pi_P^{i,j}$ is party P j-th authentication session following the completed registration session $\pi_P^{i,0}$. Thus, each party P is associated with a set of session oracles $\{\pi_P^{i,j}\}_{\langle i,j\rangle \in \mathbb{N}_0^2}$ that for ease of notation we denote simply as $\{\pi_P^{i,j}\}$. Moreover, at the beginning of each time period $\omega_I^\ell \in \Omega_I$ all pending sessions are terminated by the Refresh procedure. Thus, each session oracle operates in and is associated with exactly one time period $\omega_I^\ell \in \Omega_I$. All session oracles of party P in time period ω_I^ℓ share the same storage.

Session identifiers. The protocol implementation must define a *session identifier* sid_{SPC} that allows to uniquely identify the session between an authenticator oracle and a set of identity server oracles. To uniquely identify a session, the sid_{SPC} can be defined as a function of the protocol transcript including messages of each participant that are unique among all possible protocol executions with an overwhelming probability.

Partnering. We extend the partnering notion of [16]. In our context, partnering captures the intuitive idea that an *honest* server session that outputs an accept vote, has accepted a conversation with an authenticator session of an authenticator that previously registered at that same server. More formally, we say that authenticator oracle $\pi_T^{i,j}$ and server oracle $\pi_S^{q,l}$ are partnered if both oracles accept and: i) S is honest; ii) if $j = 0 \wedge l = 0$, they share the same sid_{SPC}; iii) if $j > 0 \wedge l > 0$, they share the same sid_{SPC} and $\pi_T^{i,0}$ and $\pi_S^{q,0}$ are partnered. In our distributed scenario, partnership is a surjective relation that maps a set of identity server oracles to an authenticator oracle.

Security experiment 1 (Survivable passwordless challenge response)**.** *The*

security experiment is run between a challenger and an adversary $\mathcal{A}$. At the beginning of the experiment the challenger defines the set of identity providers $\mathcal{I}$, the set of authenticators $\mathcal{T}$, the set of identity servers $\mathcal{S}$, and identities id_I for all $I \in \mathcal{I}$ and id_S for all $S \in \mathcal{S}$. The challenger generates authenticator key pairs $\langle ak_T, vk_T \rangle \leftarrow \mathsf{Spc.Kgen}()$ for all authenticators $T \in \mathcal{T}$ and gives each authenticator public key vk_T to $\mathcal{A}$ and to server oracles. The challenger also defines and gives $\mathcal{A}$ public values n_I, k_I for all $I \in \mathcal{I}$. The adversary chooses a target identity provider $I_t \in \mathcal{I}$ and sends I_t to the challenger. All remaining identity providers are under the adversary control, i.e. servers $S \in I \in \mathcal{I} \setminus I_t$ are corrupt*. After the setup phase, the security experiment proceeds in a series of rounds. The adversary chooses when to terminate a round and proceed to the next. At the beginning of round ℓ adversary $\mathcal{A}$ chooses a subset $\Sigma^\ell \subset \mathcal{S}_{I_t}$ of at most k_I identity servers of the target identity provider I_t, and gives Σ^ℓ to the challenger. Identity servers in Σ^ℓ are* corrupt*, while the others ($\mathcal{S}_{I_t} \setminus \Sigma^\ell$) are* honest*. The adversary may set the server registration contexts $\{\mathsf{rcs}_S\}_{S \in \Sigma^\ell}$ of corrupt servers to arbitrary values. At the beginning of each round the challenger executes the* Spc.Refresh *algorithm. In each round, $\mathcal{A}$ can interact with authenticators in $\mathcal{T}$ and identity servers in $\mathcal{S}$ via the following queries:*

- $\mathsf{Start}(\pi_S^{i,j})$*: the challenger instructs server oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.rchallenge}(\mathsf{id}_S)$ *if $j = 0$ or* $\mathsf{Spc.achallenge}(\mathsf{id}_S)$ *if $j > 0$. The resulting challenge is returned to $\mathcal{A}$.*

- $\mathsf{Challenge}(\pi_T^{i,j}, M)$*: the challenger instructs authenticator oracle $\pi_T^{i,j}$ to execute* $\mathsf{Spc.rresponse}(ak_T, \{\mathsf{rct}_T\}, M)$ *if $j = 0$ or* $\mathsf{Spc.aresponse}(\{\mathsf{rct}_T\}, M)$ *if $j > 0$ with the given command M. The resulting response is returned to $\mathcal{A}$.*

- $\mathsf{Clone}(T)$*: the challenger marks T as* cloned*, adds a new authenticator T' to $\mathcal{T}$, marks T' as* cloned*, sets T' internal state equal to T internal state ($\langle ak_T, vk_T \rangle$, $\{\mathsf{rct}_T\}$), and returns the internal state to $\mathcal{A}$.*

- $\mathsf{Complete}(\pi_S^{i,j}, T, R, cc, auid)$*: if T is cloned then the challenger extracts all challenges $\langle ch \rangle$ in cc and then executes the following: if $j = 0$ it computes $\langle M_r, cc', auid' \rangle \leftarrow \mathsf{Spc.rcommand}(\mathsf{id}_I, \langle ch \rangle)$, instructs oracle $\pi_{T'}^{i,j}$ for all clones T' of T to execute $\langle R_r, \{\mathsf{rct}_{T'}\}' \rangle \leftarrow \mathsf{Spc.rresponse}(ak_{T'}, \{\mathsf{rct}_{T'}\}, M_r)$ and instructs oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_{T'}, rc_S, R_r, cc, auid)$*; if $j > 0$ it computes $\langle M_a, cc' \rangle \leftarrow \mathsf{Spc.acommand}(\mathsf{id}_I, \langle ch \rangle)$, instructs oracle $\pi_{T'}^{i,j}$ for all clones T' of T to execute $\langle R_a, \{rct_{T'}\}' \rangle \leftarrow$* $\mathsf{Spc.aresponse}(\{\mathsf{rct}_{T'}\}, M_a)$ *and instructs oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R_a, cc)$*. Then, in any case ($T$ cloned or not cloned), the challenger instructs server oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_T, rc_S, R, cc, auid)$ *if $j = 0$, or* $\mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R, cc)$ *if $j > 0$; the result is given to $\mathcal{A}$.*

The adversary wins the experiment if an honest *identity server oracle outputs* accept *but it is not uniquely partnered with an authenticator oracle or if there*

exists distinct disjoint subsets of $|Q|_{min} - k$ honest *identity server oracles that output* accept *and are partnered with the same authenticator oracle.*

We note that although the ability of the challenger to extract all individual challenges from a collective challenge may restrict the validity of the security experiment to a certain number of specifications of Spc.rcommand and Spc.acommand where public extraction is possible, we argue that in practice this is not a major limitation because *cc* can be specified with data structures that allow public extraction (e.g., an array of challenges). The complexity of the Complete query is meant to prevent trivial ways of winning the game with zero partnered authenticator oracles if the adversary clones an authenticator.

Definition 4.5.1 (SPC advantage)**.** *Let* Π *be a survivable passwordless challenge-response protocol. We define* $\mathsf{Adv}^{\mathrm{spc}}_{\Pi}(\mathcal{A})$ *as the probability that adversary* $\mathcal{A}$ *wins the security experiment.*

Definition 4.5.2 (Secure SPC)**.** *A survivable passwordless challenge-response protocol* Π *is secure if the quantity* $\mathsf{Adv}^{\mathrm{spc}}_{\Pi}(\mathcal{A})$ *is negligible.*

We note that for Definition 4.5.2 to hold, *all* honest servers oracles that output an accept vote must have a unique authenticator partner. However, an authenticator oracle can be partnered with multiple server oracles. Moreover, we note that Definition 4.5.2 does not require the decisions of honest identity servers in Q to be consistent. With Definition 4.5.2 we guarantee that in a secure SPC protocol, in all possible identity servers subsets $Q \subseteq \mathcal{S}_I : |Q| \geq |Q|_{min}$, there is always at least an honest identity server that rejects authentication sessions from unregistered or cloned authenticators. To this aim, a secure SPC protocol specification in the sense of Definition 4.5.2 must define the bounds of public values $|Q|_{min}$ and n.

4.5.3 Survivable WebAuthn (SWA) specification

We describe and discuss the security of Survivable WebAuthn (SWA): a WebAuthn-compliant SPC protocol specification. We detail the SWA specifications of the Register and Authenticate subprotocols and related subroutines in Figures 4.4 and 4.5. We now describe the most relevant design choices and data structures. We define SWA *authenticator registration contexts* as $\mathsf{rct}_T = \langle \mathsf{id}_I, auid, cid, sk, sc \rangle$, where id_I denotes the associated identity provider which acts as scope, *cid* and *auid* denote the identifiers of the credential and of the user that owns it, *sk* denotes the secret key of the public key credential, and *sc* denotes the signature counter incremented each time *sk* is used for signing. We define SWA *server registration contexts* as $\mathsf{rcs}_S = \langle auid, cid, pk, sc, cc, \sigma \rangle$. Notations *auid*, *cid* and *sc* preserve the same semantics of authenticator registration contexts. Notation *pk* denotes the public key of the credential, *cc* and σ denote the latest collective challenge and latest command attestation signature received by the server (see Sections 4.4 and 4.5.1). For compatibility with FIDO2 authenticators, the SWA authenticator registration are identical to FIDO2-WebAuthn authenticator registration contexts. Instead, we extend FIDO2-WebAuthn server

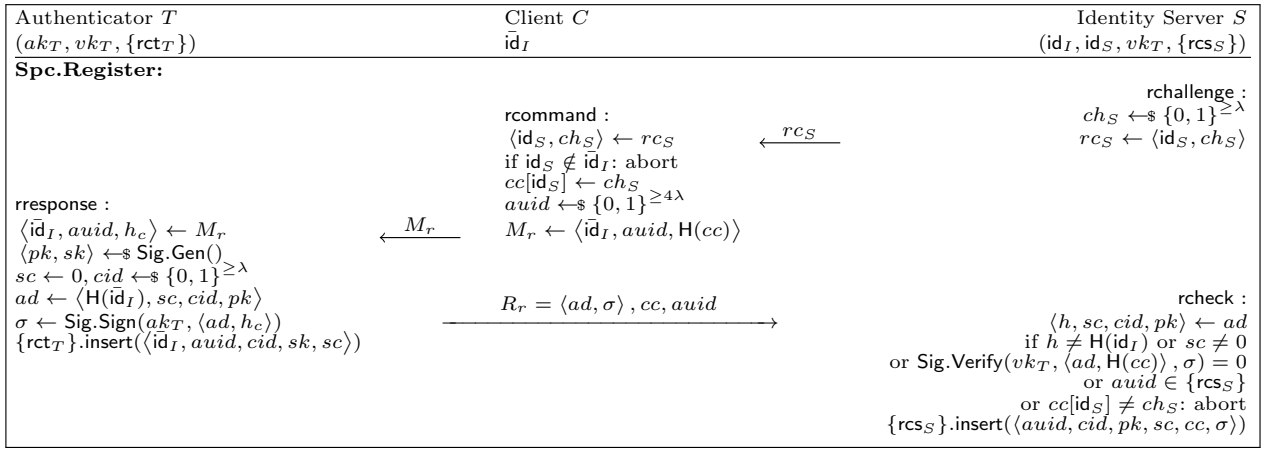

Figure 4.4: SWA specification of the SPC Register subprotocol

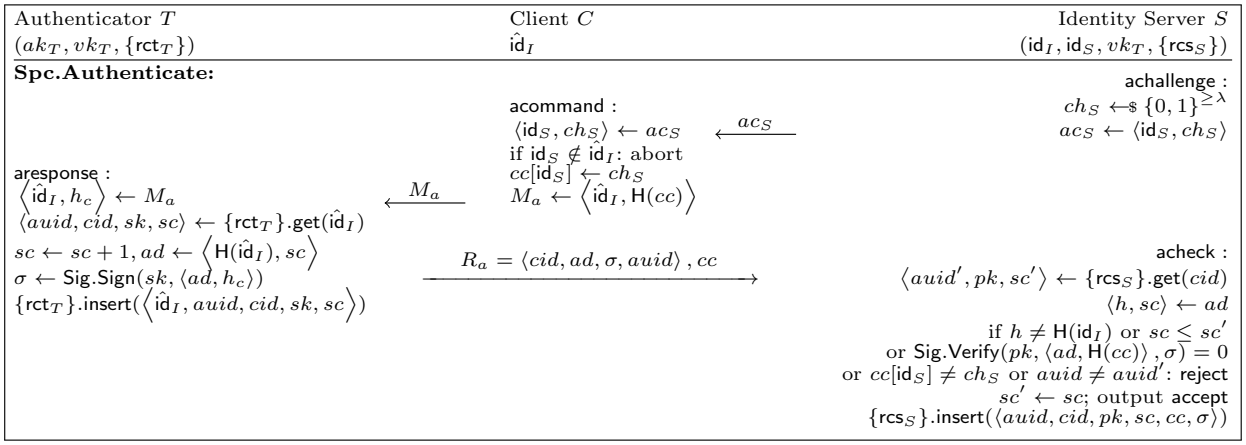

Figure 4.5: SWA specification of the SPC Authenticate subprotocol

$\mathsf{Spc.Refresh}(\mathsf{id}_I, k_I, \{\mathsf{rcs}_{S_1}\}, \ldots, \{\mathsf{rcs}_{S_{n_I}}\})$

1 : $\forall i \in [n_I] : R_{S_i} \leftarrow \{\langle auid, cid, pk\rangle : \langle auid, cid, pk, sc, cc, \sigma\rangle \in \{\mathsf{rcs}_{S_i}\}\}$

2 : $O \leftarrow \biguplus_{i\in[n_I]} R_{S_i}, \quad G \leftarrow \bigcup_{i\in[n_I]} \{\mathsf{rcs}_{S_i}\}$

3 : $V \leftarrow \{v : v \in O \wedge \|v\| \geq (k_I + 1)\}$

4 : $W \leftarrow \{\langle auid, cid, pk, sc, cc, \sigma\rangle \in G : \langle auid, cid, pk\rangle \in V \wedge$
$((sc = 0 \wedge \mathsf{Sig.Verify}(pk, \langle \mathsf{H}(\mathsf{id}_I), sc, cid, pk\rangle, \mathsf{H}(cc)\rangle, \sigma) = 1)$
$\vee (sc > 0 \wedge \mathsf{Sig.Verify}(pk, \langle \mathsf{H}(\mathsf{id}_I), sc\rangle, \mathsf{H}(cc)\rangle, \sigma) = 1))\}$

5 : $W' \leftarrow \{\langle auid, cid, pk, sc, cc, \sigma\rangle \in W :$
$sc = \mathsf{max}(\{sc' : \langle auid', cid', pk', sc', cc', \sigma'\rangle \in W \wedge \langle auid, cid, pk\rangle = \langle auid', cid', pk'\rangle\})\}$

6 : **return** W'

Figure 4.6: SWA specification of the SPC Refresh subprotocol

registration contexts by including cc and σ that are used in the Refresh subprotocol. *Registration and authentication challenges* rc_S and ac_S are identical to FIDO2-WebAuthn challenges, and defined as $\langle \mathsf{id}_S, ch_S\rangle$, where ch_S is a uniformly sampled λ-bit string and id_S is the identity of the server that generates the challenges. *Registration and authentication commands* M_r and M_a are syntactically identical to FIDO2-WebAuthn commands but semantically different, as the identity included in the command for scoping the credential is that of the identity provider that owns the servers (instead of that of the server itself as in FIDO2-WebAuthn). As in FIDO2-WebAuthn, the identity information is that observed by the client ($\bar{\mathsf{id}}_I$ for registration and $\hat{\mathsf{id}}_I$ for authentication). We define *collective challenge* cc as an associative array which uses identity servers identities id_S as keys to access the individual servers challenges (rc_S for registration and ac_S for authentication). Finally, for compatibility with FIDO2 authenticators, the registration and authentication *command attestations* R_r and R_a are identical to FIDO2-WebAuthn command attestations.

Refresh. Figure 4.6 shows the SWA specification of the Spc.Refresh subprotocol, executed by identity provider I at the beginning of each time period ω_I^ℓ. First, for each set of server registration contexts $\{\mathsf{rcs}_S\}$, the protocol removes possible duplicate credentials $\langle auid, cid, pk\rangle$ (Line 1). Second, it creates a credential multiset that includes the registration contexts of all identity servers (Line 2) to compute the set of credentials with multiplicity at least equal to $(k_I + 1)$ (Line 3, where $||v||$ denotes multiplicity of v). Third, it computes the set W of authentic registration contexts (i.e. satisfying Sig.Verify, Line 4) with sufficient multiplicity ($\langle auid, cid, pk\rangle \in V$). Finally, it returns the set W' of authentic registration contexts with the highest signature counter sc (Lines 5 and 6). The identity provider can reset the registration contexts of all identity servers with set W'.

Security. We take a modular approach to demonstrate the security of the proposed SWA protocol. We first notice that the security of Spc.Refresh is a necessary condition for the security of the proposed protocol. In fact, an adversary may abuse Spc.Refresh to gain write privileges to the registration contexts of honest identity servers it does not control. For example, this may

be exploited to set signature counters on honest identity servers to evade clone detection. We define the security of $\mathsf{Spc.Refresh}$ in terms of Experiment 2, prove that the proposed SWA specification of $\mathsf{Spc.Refresh}$ is secure in Lemma 1 and finally prove the security of SWA in Theorem 1.

Security experiment 2 (SPC Refresh)**.** *The security experiment is run between a challenger and adversary $\mathcal{A}$. The challenger defines identity id_I and values $n, k_I, \ell \in \mathbb{N}$, computes $\langle sk, pk\rangle \leftarrow\!\!\$\ \mathsf{Sig.Gen}()$, and builds an $n \times \ell$ matrix $\mathcal{M}$, where $sc^i_j \in \mathbb{N}$ for $i \in [n]$, $j \in [\ell]$ such that $sc^i_{j-1} \leq sc^i_j$, $auid \leftarrow\!\!\$\ \{0,1\}^{\geq\lambda}$, $cid \leftarrow\!\!\$\ \{0,1\}^{\geq\lambda}$, $cc_j \leftarrow\!\!\$\ \{0,1\}^{\geq n\lambda}$, and $\sigma^i_1 \leftarrow\!\!\$\ \mathsf{Sig.Sign}(sk, \langle \mathsf{H}(\mathsf{id}_I), sc^i_1, cid, pk, \mathsf{H}(cc_1)\rangle)$ and $\sigma^i_j \leftarrow\!\!\$\ \mathsf{Sig.Sign}(sk, \langle \mathsf{H}(\mathsf{id}_I), sc^i_j, \mathsf{H}(cc_j)\rangle)$ for $j > 1$.*

$$\mathcal{M} = \begin{pmatrix} (auid, cid, pk, sc^1_1, cc_1, \sigma^1_1) & \ldots & (auid, cid, pk, sc^1_\ell, cc_\ell, \sigma^1_\ell) \\ \ldots & \ldots & \ldots \\ (auid, cid, pk, sc^n_1, cc_1, \sigma^n_1) & \ldots & (auid, cid, pk, sc^n_\ell, cc_\ell, \sigma^n_\ell) \end{pmatrix}$$

The challenger then gives $\mathcal{A}$ value k_I and matrix $\mathcal{M}$. $\mathcal{A}$ returns to the challenger set $\Sigma \subset [n] : |\Sigma| \leq k$, and a vector $\mathbf{m}$ such that $\mathbf{m}_i = \mathcal{M}_{i,\ell}$ for $i \notin \Sigma$. The challenger executes $\mathbf{v} \leftarrow \mathsf{Spc.Refresh}(\mathsf{id}_I, k_I, \{\mathbf{m}_1\}, \ldots, \{\mathbf{m}_n\})$ and returns $\mathbf{v}$. Let $\left\langle \hat{auid}_i, \hat{cid}_i, \hat{pk}_i, \hat{sc}_i, \hat{cc}_i, \hat{\sigma}_i \right\rangle = \mathbf{v}_i$. The adversary wins the experiment if $\hat{auid}_i \neq auid$ or $\hat{cid}_i \neq cid$ or $\hat{pk}_i \neq pk$, or $\hat{sc}_i < sc^i_\ell$ for $i \notin \Sigma$ and $\mathsf{Sig.Verify}(\hat{\sigma}_i, \langle \mathsf{H}(\mathsf{id}_I), \hat{sc}_i, \mathsf{H}(\hat{cc}_i)\rangle, \hat{pk}_i) = 1$, or $\hat{sc}_i > \max_{h \notin \Sigma}(sc^h_\ell)$ and $\mathsf{Sig.Verify}(\hat{\sigma}_i, \langle \mathsf{H}(\mathsf{id}_I), \hat{sc}_i, \mathsf{H}(\hat{cc}_i)\rangle, \hat{pk}_i) = 1$.

Experiment 2 captures an adversary trying to swap or overwrite registered credentials, trying to decrement signature counters on honest identity servers to evade clone detection after time period ω^ℓ_I, or trying to increment signature counters on honest identity servers to introduce possible false positives in clone detection.

Definition 4.5.3 (SPC Refresh advantage)**.** *Let Π be a SPC Refresh protocol. We define $\mathsf{Adv}^{\mathsf{spc.refresh}}_{\Pi}(\mathcal{A})$ as the probability that adversary $\mathcal{A}$ wins the security experiment.*

Definition 4.5.4 (Secure SPC Refresh)**.** *A SPC Refresh protocol Π is secure if the quantity $\mathsf{Adv}^{\mathsf{spc.refresh}}_{\Pi}(\mathcal{A})$ is negligible.*

Lemma 1 (Secure SWA Refresh)**.** *If the amount of malicious identity servers $k_I < \lceil n_I/2 \rceil$ for any time period $\omega^\ell_I \in \Omega_I$, for any adversary $\mathcal{A}$ against the $\mathsf{Spc.Refresh}$ protocol, there exists an adversary $\mathcal{D}$ such that:*

$$\mathsf{Adv}^{\mathsf{spc.refresh}}_{\mathsf{swa.refresh}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{euf-cma}}_{\mathsf{Sig}}(\mathcal{D})$$

The proposed SWA protocol adopts $\langle ad, \mathsf{H}(cc)\rangle$ as session identifier. In Section 4.9 we prove Lemma 1 and the following theorem, which shows that if H is a collision-resistant hash function [43], Sig is an existentially unforgeable signature scheme [61], SWA $\mathsf{Refresh}$ is a secure SPC Refresh protocol, and

$|Q|_{min} = 2k_I + 1$, $n_I \in [2k_I + 1, 3k_I + 1]$ then the proposed SWA protocol is a secure SPC protocol in the sense of Definition 4.5.2.

Theorem 1 (Survivable WebAuthn security)**.** *If the SWA Refresh specification is a secure SPC Refresh protocol and* $|Q|_{min} = 2k_I + 1$, $n_I \in [2k_I + 1, 3k_I + 1]$ *then, for any efficient adversary* $\mathcal{A}$ *against the survivable WebAuthn protocol that makes at most* η *queries to* Start *and* θ *queries to* Complete*, there exist efficient adversaries* $\mathcal{B}$*,* $\mathcal{D}$ *such that:*

$$\mathsf{Adv}^{\mathrm{spc}}_{\mathrm{SWA}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{B}) + \theta \cdot \mathsf{Adv}^{\mathsf{euf-cma}}_{\mathsf{Sig}}(\mathcal{D}) + (\eta^2 + \theta^2) \cdot 2^{-\lambda}$$

As Theorem 1 shows, one can reduce the security of the SWA protocol to collision-resistance of hash function H and existential unforgeability under chosen message attack of signature scheme Sig.

4.6 Survivable Passwordless SSO (SPS) protocol

4.6.1 SPS operations framework

A SPS protocol consists of six subprotocols: *authenticator key generation*, *identity provider setup*, *identity server key generation*, *register*, *authenticate*, *refresh*.

Authenticator Key Generation

$\langle ak_T, vk_T \rangle \leftarrow\!\!\$\ \mathsf{Sps.Akgen}()$: executed once by authenticator T, returns the authenticator key pair $\langle ak_T, vk_T \rangle$. It coincides with the Spc.Kgen routine of SPC.

Identity provider Setup

$\langle k_I, n_I, \Omega_I \rangle \leftarrow \mathsf{Sps.Ppgen}()$: executed once for each identity provider $I \in \mathcal{I}$, returns security threshold k_I, number of identity servers n_I and the set of time periods Ω_I.

Identity Server Key Generation

$\langle sk_S, pk_S \rangle_{S \in \mathcal{S}_I} \leftarrow\!\!\$\ \mathsf{Sps.Skgen}(\omega_I^\ell, n_I)$: executed by identity provider I, returns key pairs $\langle sk_S, pk_S \rangle_{S \in \mathcal{S}_I}$ (one key pair for each identity server S owned by I), given a time period $\omega_I^\ell \in \Omega_I$ and the number of identity servers n_I.

Register (Sps.Register): executed among client C, authenticator T, and the set of all identity servers $\mathcal{S}_I$ of identity provider I. The parties first execute Spc.Register. Then, after executing the last Spc.rcheck subroutine, all identity servers (which know user identity *uid* that is controlling the client, see Section 4.4) also execute an additional routine:

- $im' \leftarrow \mathsf{Sps.idbind}(im, auid, uid)$: executed by each identity server, takes as input identity map im, user identity uid and authenticator user identifier $auid$ (returned by Spc.command). It returns the updated identity map im' which binds uid to $auid$;

Authenticate (Sps.Authenticate): allows service provider V to verify the outcome of a SPC authentication session among a client C, authenticator T, and a subset of identity servers $Q \subseteq \mathcal{S}_I$. We denote the identity provider

intended identities for the service provider as $\tilde{\mathsf{id}}_I$, for the client as $\hat{\mathsf{id}}_I$, and the service provider intended identity for each identity server as $\bar{\mathsf{id}}_V$. Authenticate consists of the following subprotocols:

- $\left\langle \mathsf{id}_V, \left\langle \tilde{\mathsf{id}}_S, a_S \right\rangle_{S \in \mathcal{S}_I} \right\rangle \leftarrow \mathsf{Sps.abegin}(\tilde{\mathsf{id}}_I)$: executed by service provider V, returns collective session information $\left\langle \tilde{\mathsf{id}}_S, a_S \right\rangle_{S \in \mathcal{S}_I}$, where a_S denotes session information for identity server $\tilde{\mathsf{id}}_S \in \tilde{\mathsf{id}}_I$, given intended identity $\tilde{\mathsf{id}}_I$.
- $\langle b_S, \{\perp \mid R_a\} \rangle \leftarrow \mathsf{Spc.Authenticate}(\hat{\mathsf{id}}_I, \left\langle \hat{\mathsf{id}}_S \right\rangle_{S \in Q})$: client C uses identity servers $\left\langle \hat{\mathsf{id}}_S \right\rangle_{S \in Q}$ owned by $\hat{\mathsf{id}}_I$ to execute the SPC Authenticate subprotocol. At the end of the protocol execution, each server obtains vote b_S and R_a (returned by $\mathsf{Spc.acheck}$ and $\mathsf{Spc.aresponse}$): if b_S is accept, R_a is used in the next routine, $\perp$ otherwise.
- $\{\perp, v_S\} \leftarrow \mathsf{Sps.release}(\mathsf{id}_S, \bar{\mathsf{id}}_V, R_a, sk_S, a_S, b_S, im)$: executed by each identity server S, if $b_S = \mathsf{reject}$ it returns $\perp$, otherwise it returns identity attestation v_S of the authenticated user identity, authenticated by secret key sk_S, given identities id_S and $\bar{\mathsf{id}}_V$, authentication response R_a, the identity server signing key sk_S, session information a_S and identity map im.
- $\{\perp, c_V\} \leftarrow \mathsf{Sps.count}(k_I, \mathsf{id}_V, \tilde{\mathsf{id}}_I, \langle v_S, pk_S, a_S \rangle_{S \in Q})$: executed by service provider V, returns session cookie c_V if enough identity attestations $\langle v_S \rangle_{S \in Q}$ are considered valid, otherwise $\perp$, given security threshold k_I, service provider identity id_V, identity provider intended identity $\tilde{\mathsf{id}}_I$, and public keys and session information $\langle pk_S, a_S \rangle_{S \in Q}$. If count returns c_V we say the service provider has *accepted* the authentication for the user identity in v_S.

Refresh ($\mathsf{Sps.Refresh}$): executed by identity provider I at the beginning of each time period $\omega_I^\ell \in \Omega_I$ on all of its identity servers $S \in \mathcal{S}_I$. It terminates pending sessions and restores possibly corrupt identity servers to a safe state. Refresh consists of two routines. First, a trivial extension of $\mathsf{Spc.Refresh}$ which we omit for brevity, to restore possibly corrupt registration contexts and identity maps for all identity servers $\mathcal{S}_I$; $\mathsf{Sps.Skgen}$ to generate new keys for all identity servers $\langle sk_S, pk_S \rangle_{S \in \mathcal{S}_I}$, given time period ω_I^ℓ and the number of identity servers n_I.

Intuitively, SPS correctness requires that a client and a service provider always share the same secret session cookie created as a result of an authentication that is voted by enough identity servers of an identity provider, and which is consistent with a prior registration at the identity servers of the same identity provider, if and only if the client collected all identity attestations from identity servers belonging to the same identity provider required by the service provider. To formally capture the correctness requirement of service provider and client sharing the same session cookie at the end of a correct SPS authentication, we

introduce a match function. The match function serves as an abstraction for the correctness definition and does not model the behavior of any protocol participant. At the end of the protocol the client inputs the received session cookie $\hat{c_V}$ and the service provider inputs the generated session cookie c_V. The function $b \leftarrow \mathsf{match}(\hat{c_V}, c_V)$ returns $b = 1$ if $\hat{c_V} = c_V$, $b = 0$ otherwise.

Correctness (SPS). Correctness requires that for any identity provider identities id_I, $\hat{\mathsf{id}}_I$, $\bar{\mathsf{id}}_I$, $\tilde{\mathsf{id}}_I$, service provider identities id_V, $\bar{\mathsf{id}}_V$, server identity id_S, and user identity uid, for all $k_I, n_I \in \mathbb{N}$ such that $k_I < n_I$, for all $S \in Q \subseteq \mathcal{S}_I$, every distribution D_λ over $\{0,1\}^{\rho(\lambda)}$, and for all secret session cookies $\hat{c_V}, c_V \in \{0,1\}^{\rho(\lambda)}$ for some polynomial $\rho(\lambda)$ and security parameter $\lambda \in \mathbb{N}$ if:

$$
\begin{array}{l}
\langle k_I, n_I, \Omega_I \rangle \leftarrow \mathsf{Sps.Ppgen}() \\
\langle ak_T, vk_T \rangle \leftarrow\!\!\$\ \mathsf{Sps.Akgen}() \\
im \leftarrow \{\} \\
\langle sk_{S_i}, pk_{S_i} \rangle_{i \in [n_I]} \leftarrow\!\!\$\ \mathsf{Sps.Skgen}(\omega_I^\ell, n_I) \\
rc_{S_i} \leftarrow\!\!\$\ \mathsf{Spc.rchallenge}(\mathsf{id}_{S_i}), \forall i \in [n_I] \\
\langle M_r, cc, auid \rangle \leftarrow \mathsf{Spc.rcommand}(\bar{\mathsf{id}}_I, \langle rc_{S_i} \rangle_{i \in [n_I]}) \\
\langle R_r, \{\mathsf{rct}_T\}' \rangle \leftarrow\!\!\$\ \mathsf{Spc.rresponse}(ak_T, \{\mathsf{rct}_T\}, M_r) \\
\{\mathsf{rcs}_{S_i}\}' \leftarrow \mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_{S_i}, \{\mathsf{rcs}_{S_i}\}, vk_T, rc_{S_i}, R_r, cc, auid), \forall i \in [n_I] \\
im' \leftarrow \mathsf{Sps.idbind}(im, auid, uid) \\
\langle \langle sk_{S_i}, pk_{S_i} \rangle_{i \in [n_I]}, \{\mathsf{rcs}\}' \rangle \leftarrow \mathsf{Sps.Refresh}(\mathsf{id}_I, k_I, \omega_I^\ell, \{\mathsf{rcs}_{S_1}\}', \ldots, \{\mathsf{rcs}_{S_{n_I}}\}') \\
\{\mathsf{rcs}_{S_i}\}'' \leftarrow \{\mathsf{rcs}\}', \forall i \in [n_I] \\
\langle \mathsf{id}_V, \langle \tilde{\mathsf{id}}_{S_i}, a_{S_i} \rangle_{i \in [|Q|]} \rangle \leftarrow \mathsf{Spc.abegin}(\tilde{\mathsf{id}}_I) \\
ac_{S_i} \leftarrow\!\!\$\ \mathsf{Spc.achallenge}(\mathsf{id}_{S_i}), \forall i \in [|Q|] \\
\langle M_a, cc \rangle \leftarrow \mathsf{Spc.acommand}(\bar{\mathsf{id}}_I, \langle ac_{S_i} \rangle_{i \in [|Q|]}) \\
\langle R_a, \{\mathsf{rct}_T\}'' \rangle \leftarrow\!\!\$\ \mathsf{Spc.aresponse}(\{\mathsf{rct}_T\}', M_a) \\
\langle b_{S_i}, \{\mathsf{rcs}_{S_i}\}''' \rangle \leftarrow \mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_{S_i}, \{\mathsf{rcs}_{S_i}\}'', ac_{S_i}, R_a, cc), \forall i \in [|Q|] \\
v_{S_i} \leftarrow \mathsf{Sps.release}(\mathsf{id}_{S_i}, \bar{\mathsf{id}}_V, R_a, sk_{S_i}, a_{S_i}, b_{S_i}, im'), \forall i \in [|Q|] \\
c_V \leftarrow \mathsf{Sps.count}(k_I, \mathsf{id}_V, \tilde{\mathsf{id}}_I, \langle v_{S_i}, pk_{S_i}, a_{S_i} \rangle_{i \in [|Q|]}) \\
b \leftarrow \mathsf{match}(\hat{c_V}, c_V)
\end{array}
$$

then condition:

$$b = (c_V \overset{?}{\sim} D_\lambda) \wedge (\mathsf{id}_I \overset{?}{=} \tilde{\mathsf{id}}_I) \wedge (\mathsf{id}_I \overset{?}{=} \bar{\mathsf{id}}_I) \wedge (\mathsf{id}_I \overset{?}{=} \hat{\mathsf{id}}_I) \bigwedge_{i \in [|Q|]} (\mathsf{id}_{S_i} \in \mathsf{id}_I)$$

where $c_V \overset{?}{\sim} D_\lambda$ is true if c_V is distributed according to D_λ, holds with probability 1 for all time periods $\omega_I^\ell \in \Omega_I$.

4.6.2 SPS security model

Trust assumptions. The trust assumptions of the SPS protocol include the same assumptions adopted for the SPC protocol (see Section 4.5.2), with additional ones due to the new actors and communication channels. First, we require server-to-client authenticated communication channels. As in [16], this is done to guarantee to the user and the client the true identity of identity servers during registration, to enforce credential scope to a single identity provider and prevent reusing a credential to authenticate at different identity providers. Moreover, we require confidential and authenticated channels between the client and the service provider, otherwise the attacker can trivially impersonate a user. We

assume that the client is honest, that the authenticator is tamper-proof (the adversary has read-only access to the authenticator internal state), and that communication channels between the authenticator and the client are mutually authenticated. In Section 4.11 we prove that the composition with the CTAP2 protocol allows to drop the latter assumption and still obtain a secure protocol.

Session oracles. The protocol is executed by session oracles of party $P \in \mathcal{T} \cup \mathcal{S} \cup \mathcal{V} \cup \mathcal{C}$. Session oracle $\pi_P^{i,j}$ when $P \in \mathcal{T} \cup \mathcal{S}$ maintains the same semantics of the SPC protocol. Session oracle π_V^i for $V \in \mathcal{V}$ and π_C^i for $C \in \mathcal{C}$ is the i-th pending or completed authentication session of service provider V and client C respectively.

Session identifiers. The protocol must define a global session identifier sid_{SPS} to uniquely identify an authentication session between a service provider, a client, an authenticator and identity servers. Moreover, the protocol must define a subsession identifier ssid_{SPS}, to uniquely identify individual identity server sessions that are created to participate in the global session sid_{SPS}. If ssid_{SPS} is a subsession of sid_{SPS} we say that ssid_{SPS} *belongs* to sid_{SPS}, and we write $\mathsf{ssid}_{SPS} \in \mathsf{sid}_{SPS}$.

SPS subsession attestation binding. To prevent *shared session attacks* (see Section 4.4) the protocol must bind an identity attestation to the subsession ssid_{SPS} of the identity server that released the attestation. This binding acts as a scope that allows to detect attestations injected on a different SPS authentication session.

Time period-bound attestations. To prevent *attestation mixing* attacks (see Section 4.4) the protocol must limit the scope of an identity attestation to a single time period.

SPC session-bound attestations. To guarantee *authenticator clone detection* and detect malicious interleaving of distinct SPC sessions (see Section 4.4) the protocol must bind identity attestations to the SPC authentication session.

Partnership. To capture the idea that a set of identity servers participate in independent subsessions of an authentication session initiated by a service provider, we introduce a novel notion of partnering. A service provider oracle π_V^h is partnered with an identity server oracle $\pi_S^{i,j}$ for $j > 0$ if π_V^h accepts and $\pi_S^{i,j}$ outputs an identity attestation, and both oracles share the same subsession identifier ssid_{SPS}. A service provider oracle π_V^h is partnered with a set of identity server oracles $\{\pi_S^{i,j}\}$ for $j > 0$ if π_V^h accepts in global session sid_{SPS}, π_V^h is partnered with each identity server oracle $\pi_S^{i,j}$ for $j > 0$ and the subsession identifier ssid_{SPS} of $\pi_S^{i,j}$ belongs to sid_{SPS}. Finally, we say that client oracle π_C^g and service provider oracle π_V^h are partnered if both oracles accept and share the same global session identifier sid_{SPS}.

Session cookie security. We model session cookie c_V established between client and service provider at the end of Count as a shared cryptographic key. Even if this modeling may seem too strong for data meant to be used as a browser cookie, it allows us to adopt well-studied models and provide the strongest security guarantees with known efficient and practical implementations. As

a result, we require the protocol to guarantee *freshness* and *confidentiality* of session cookie, as in key transport protocols [33]. We model freshness by allowing the adversary to ask session cookie to partnered client or service provider oracles. Session cookies and oracles asked by the adversary in this manner are said *unfresh*, otherwise they are *fresh*. We model confidentiality with the following well-known indistinguishability game [20, 19, 33]. Let $\lambda \in \mathbb{N}$ be a security parameter and D_λ over $\{0,1\}^{\rho(\lambda)}$ for some polynomial $\rho(\lambda)$ be the distribution from which a target fresh session cookie c_V is drawn. A fair coin is flipped. If it lands heads, then c_V is returned to the adversary. If it lands tails, a random element from D_λ is returned. The adversary must guess whether it was given the target c_V or a random element from D_λ. We denote the probability that adversary $\mathcal{A}$ wins the indistinguishability game as $\mathsf{Adv}_{\mathrm{IND}}(\mathcal{A})$.

SPS security guarantees. As anticipated in Section 4.4 we extend the definitions of *authentication* and *session integrity* of [51] to our distributed scenario, and we denote them as *SPS authentication* and *SPS session integrity*. Intuitively:

- *SPS authentication* captures that, in any time period, even if an adversary has compromised up to k_I identity servers of an honest identity provider I, the adversary learns nothing about the established session cookie between client and service provider, nor can evade cloned authenticator detection;
- *SPS session integrity* captures that, in any time period, even if an adversary has compromised up to k_I identity servers of an honest identity provider I, an honest service provider does not accept an authentication by counting unsolicited attestations from identity servers of identity provider I, and an honest service provider does not accept a user identity uid' if the user authenticated to identity servers for a different user identity uid.

We propose Definitions 4.6.1 and 4.6.2 for a formal treatment.

Definition 4.6.1 (SPS Authentication)**.** *A SPS protocol* Π *guarantees secure authentication if when fresh service provider oracle* π_V^h *accepts then (i)* π_V^h *is partnered with at least* $|Q|_{min}$ *identity server oracles* $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_I}$ *of distinct identity servers* S *of the same identity provider* I *that are uniquely partnered with the same authenticator oracle* $\pi_T^{q,l}$*, (ii) there do not exist distinct disjoint subsets of* $|Q|_{min} - k_I$ *honest identity server oracles* $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_I}$ *that output* accept *and are partnered with* $\pi_T^{q,l}$*, (iii)* π_V^h *is uniquely partnered with fresh client oracle* π_C^g*, and (iv)* $\mathsf{Adv}_{IND}(\mathcal{A})$ *is negligible.*

Definition 4.6.2 (SPS Session integrity)**.** *A SPS protocol* Π *guarantees session integrity if when service provider oracle* π_V^h *accepts then* π_V^h *has also executed* Sps.abegin*,* π_V^h *is uniquely partnered with* π_C^g*, and if* π_V^h *accepts for user identity uid, then* π_C^g *accepts for the same user identity uid and at least* $|Q|_{min}$ *identity server oracles* $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_I}$ *uniquely partnered with* π_V^h *have output* accept *for the same user identity uid.*

Security experiment 3 (SPS)**.** *The setup of this security experiment proceeds identically to Experiment 1. Moreover, adversary $\mathcal{A}$ chooses a target service provider $V_t \in \mathcal{V}$ and sends V_t to the challenger. All remaining service providers ($V \in \mathcal{V} \setminus V_t$) are under the adversary control, and the challenger marks them as* unfresh*. The experiment then proceeds in rounds as Experiment 1 with the following differences. At the beginning of round i the challenger executes the* Spc.Refresh *subprotocol and gives public keys of all identity servers $S \in \mathcal{S}_{I_t}$ to $\mathcal{A}$ and service provider oracles. At the beginning of round ℓ the challenger also gives secret keys sk_S of corrupt identity servers $S \in \Sigma^\ell$ to $\mathcal{A}$. At the end of round ℓ the challenger gives the secret keys sk_S of all identity servers $S \in \mathcal{S}_{I_t}$ to $\mathcal{A}$. During each round, the adversary can interact with authenticators, clients, identity servers and service providers in $\mathcal{T} \cup \mathcal{C} \cup \mathcal{I} \cup \mathcal{V}$ via the following queries:*

- $\mathsf{Init}(\pi_V^h, \pi_C^g, \mathsf{id})$*: the challenger instructs service provider oracle π_V^i to execute* $\mathsf{Sps.abegin}(\mathsf{id})$ *and returns all a_S values such that $S \in \Sigma^\ell$ to $\mathcal{A}$ and all a_S values to π_C^g.*

- $\mathsf{Start}(\{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}, \langle c \rangle)$*: the challenger instructs up to $n_{I_t} - k_{I_t}$ identity server oracles $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}$ of distinct servers to execute* $\mathsf{Spc.rchallenge}(\mathsf{id}_S)$ *if $j = 0$ or* $\mathsf{Spc.achallenge}(\mathsf{id}_S)$ *if $j > 0$. The generated challenges are returned to $\mathcal{A}$. W.l.o.g. either $j = 0$ or $j > 0$ for all $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}$. The challenger then executes* Spc.rcommand *if $j = 0$ or* Spc.acommand *if $j > 0$ with the returned challenges along with the tuple of k_{I_t} challenges $\langle c \rangle$ and the identities $\{\mathsf{id}_S\}$ of server oracles $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}$ and corresponding identity provider id_I. The challenger then delivers the resulting command both to $\mathcal{A}$ and $\pi_T^{q,l}$. If $l = 0$ the challenger delivers the resulting command to $\pi_T^{q,0}$ which executes* Spc.rresponse*, otherwise if $l > 0$ it delivers the command to $\pi_T^{q,l}$ which executes* Spc.aresponse *and returns the result to $\mathcal{A}$.*

- $\mathsf{Clone}(T)$*: the challenger marks T as* cloned*, adds a new authenticator T' to $\mathcal{T}$, marks T' as* cloned*, sets T' internal state equal to T internal state ($\langle ak_T, vk_T \rangle$, $\{\mathsf{rct}_T\}$), and returns the internal state to $\mathcal{A}$.*

- $\mathsf{Complete}(\pi_S^{i,j}, \pi_C^g, \mathsf{id}_V, T, R, a_S, cc, auid)$*: if T is cloned then the challenger extracts all challenges $\langle ch \rangle$ in cc and then executes the following: if $j = 0$ it computes $\langle M_r, cc', auid' \rangle \leftarrow$* $\mathsf{Spc.rcommand}(\mathsf{id}_I, \langle ch \rangle)$*, instructs oracle $\pi_{T'}^{i,j}$ for all clones T' of T to execute $\langle R_r, \{\mathsf{rct}_{T'}\}' \rangle \leftarrow$* $\mathsf{Spc.rresponse}(ak_{T'}, \{\mathsf{rct}_{T'}\}, M_r)$ *and instructs oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_{T'}, rc_S, R_r, cc, auid)$*; if $j > 0$ it computes $\langle M_a, cc' \rangle \leftarrow$* $\mathsf{Spc.acommand}(\mathsf{id}_I, \langle ch \rangle)$*, instructs oracle $\pi_{T'}^{i,j}$ for all clones T' of T to execute $\langle M_a, cc' \rangle \leftarrow$* $\mathsf{Spc.aresponse}(\{\mathsf{rct}_{T'}\}, M_a)$ *and instructs oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R_a, cc)$*. Then, in any case ($T$ cloned or not), the challenger instructs identity server oracle $\pi_S^{i,j}$ to execute* $\mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_T, rc_S, R, cc, auid)$ *if $j = 0$ or* $\mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R, cc)$ *if $j > 0$. The resulting bit b_S is returned to $\mathcal{A}$. If $j > 0$, then the challenger also instructs server oracle $\pi_S^{i,j}$ to execute* $\mathsf{Sps.release}(\mathsf{id}_S, \mathsf{id}_V, R, sk_S,$

a_S, b_S, im) *with subsession information* a_S *and returns the resulting identity attestation to* π_C^g.

- $\mathsf{Count}(\pi_V^h, \pi_C^g, \langle v \rangle, \mathsf{id}, a)$*: the challenger instructs service provider oracle* π_V^h *to execute* Sps.count *with authentication session information* a *on the tuple of identity attestations* $\langle v \rangle$ *and attestations owned by* π_C^g*, if any.* π_V^h *verifies the received attestations with public keys of identity servers owned by identity provider* id*. The challenger returns values* $\langle v \rangle$*,* a*, and the resulting session cookie* c_V *to* π_C^g.

- $\mathsf{Reveal}(\pi_P^i)$*: the challenger instructs* π_P^i *(*$P \in \mathcal{C} \cup \mathcal{V}$*) to return session cookie* c_V*, authentication session information* a *and attestations* $\langle v \rangle$*. Oracle* π_P^i*, its partner (if any), and values* c_V*,* a *and* $\langle v \rangle$ *are said* unfresh.

- $\mathsf{Test}(\pi_P^i)$*: if* π_P^i *(*$P \in \mathcal{C} \cup \mathcal{V}$*) is fresh, has accepted and owns session cookie* c_V*, then the challenger flips a fair coin* $b \leftarrow\!\!\$\ \{0,1\}$*. If* $b = 0$ *the challenger returns a random sample from* D_λ*, otherwise it instructs* π_P^i *to return* c_V*. The adversary then outputs bit* b' *and wins the indistinguishability game if* $b' = b$*.* $\mathcal{A}$ *can execute this query only once.*

The adversary wins the SPS game if conditions of Definition 4.6.1 or Definition 4.6.2 do not hold.

We note that the Start query captures authenticated and reliable communication channels from honest identity servers to the client and mutually authenticated communication channels between the client to the authenticator. Queries Init, Complete and Count model confidential and authenticated channels, which the adversary can read via the Reveal query to capture compromise of protocol participants, or the fact that the adversary may be a user of the protocol. However, the Reveal query makes oracles unfresh to prevent trivial ways of winning the game. Finally, the Reveal query also allows to capture session fixation attacks [51] which may let the adversary win the indistinguishability game and break SPS authentication.

Definition 4.6.3 (SPS advantage)**.** *Let* Π *be a SPS protocol. We define* $\mathsf{Adv}_{\Pi}^{\mathrm{sps}}(\mathcal{A})$ *as the probability that adversary* $\mathcal{A}$ *wins Experiment 3.*

Definition 4.6.4 (Secure SPS)**.** *A SPS protocol* Π *is secure if the quantity* $\mathsf{Adv}_{\Pi}^{\mathrm{sps}}(\mathcal{A})$ *is negligible.*

According to Definition 4.6.1, having a secure SPC protocol is a necessary condition to obtain a secure SPS protocol. An adversary that breaks SPC can also break SPS since there may be less than $|Q|_{min}$ uniquely partnered identity server oracles (condition i), or there may exist disjoint subsets of $|Q|_{min} - k$ honest identity servers that output accept on the same authenticator partner (condition ii). To capture the requirement of session-bound attestations, Definition 4.6.1 introduces the additional constraint of identity servers being uniquely partnered to the *same* authenticator oracle to guarantee that identity attestations correspond to the same SPC session. An important consequence of this

partnership constraint is that the adversary does not violate the security of a SPS protocol if it collects less than $|Q|_{min}$ identity attestations on distinct SPC sessions.

4.6.3 Survivable Passwordless OpenID Connect (SPOC) specification

We describe Survivable Passwordless OpenID Connect (SPOC): a specification for SPS based on OpenID Connect implicit flow [107] that extends the SWA (WebAuthn-compliant) specification. The proposal can be easily adapted with minor modifications to support other popular flows as well.

The SPOC specification of the SPS Register subprotocol is a straightforward extension of the SWA Register specification, which defines that the $\mathsf{Sps.idbind}$ routine stores the binding of *uid* to *auid* by denoting *im* as an associative array, such that $uid \leftarrow im[auid]$. Similarly, the SPOC Refresh subprotocol is a straightforward extension of the SWA Refresh specification, which restores possibly corrupt identity maps. For brevity, we omit the SPS Refresh security definition which naturally extends the SPC Refresh experiment.

We detail the SPOC specification of the SPS Authenticate subprotocol in Figure 4.7 and related subroutines $\mathsf{Sps.release}$ and $\mathsf{Sps.count}$ in Figures 4.8a and 4.8b. We focus on relevant design choices and data structures. We define SPOC *session information* $a_S = \langle st_S, nn_S \rangle$, where st_S corresponds to OIDC `state` information, and nn_S to `nonce` which are used to prevent different forms of attacks to browser sessions. Both values are uniformly sampled for each identity server for each SPS subsession to prevent *shared session attacks*. We define SPOC identity attestations $v_S = \langle \langle \langle uid, \mathsf{id}_S, \mathsf{id}_V, \mathsf{sid}_{SPC}, nn_S \rangle, \sigma \rangle, st_S \rangle$ where *uid* denotes the attested user identity, id_S denotes the attestation issuer, id_V the attestation audience, nn_S is the nonce value received within the session information a_S, σ denotes the attestation digital signature. The attestation introduces the authenticated parameter $\mathsf{sid}_{SPC} = \langle ad, \mathsf{H}(cc) \rangle$ to bind SPOC identity attestations to SWA authentication sessions and prevent *clone detection evasion* attacks. The attestation also forwards the non-authenticated value st_S, received in session information a_S.

The count algorithm shown in Figure 4.8b extends the OIDC ID token validation algorithm with the following additional checks: there must be at least $2k_I + 1$ identity attestations (Line 1), each issuer $t.\mathsf{id}_S$ must belong to the intended identity provider $\vec{\mathsf{id}}_I$ (Line 7), all attestations must have been issued as an outcome of the same Survivable WebAuthn session (Lines 11 to 15). If at least $2k_I + 1$ are valid, it returns a freshly sampled random session cookie c_V.

SPOC adopts $\langle st_S, nn_S, \mathsf{sid}_{SPC} \rangle$ as subsession identifier, where sid_{SPC} is the session identifier of the Survivable WebAuthn protocol, and adopts $\left\langle \langle st_S, nn_S \rangle_{S \in \mathcal{S}_I}, \mathsf{sid}_{SPC}, c_V \right\rangle$ as global session identifier. A subsession $\langle st_S, nn_S, \mathsf{sid}_{SPC} \rangle$ belongs to the session $\left\langle \langle st'_S, nn'_S \rangle_{S \in \mathcal{S}_I}, \mathsf{sid}'_{SPC}, c_V \right\rangle$ if $\mathsf{sid}_{SPC} = \mathsf{sid}'$ and $st_S = st'_S \wedge nn_S = nn'_S$. In Section 4.11.5 we prove the following theorem which shows that, under the same assumptions of Theorem 1, SPOC is a secure Survivable Passwordless Single Sign-On (SPS) protocol in the sense of Definition 4.6.4.

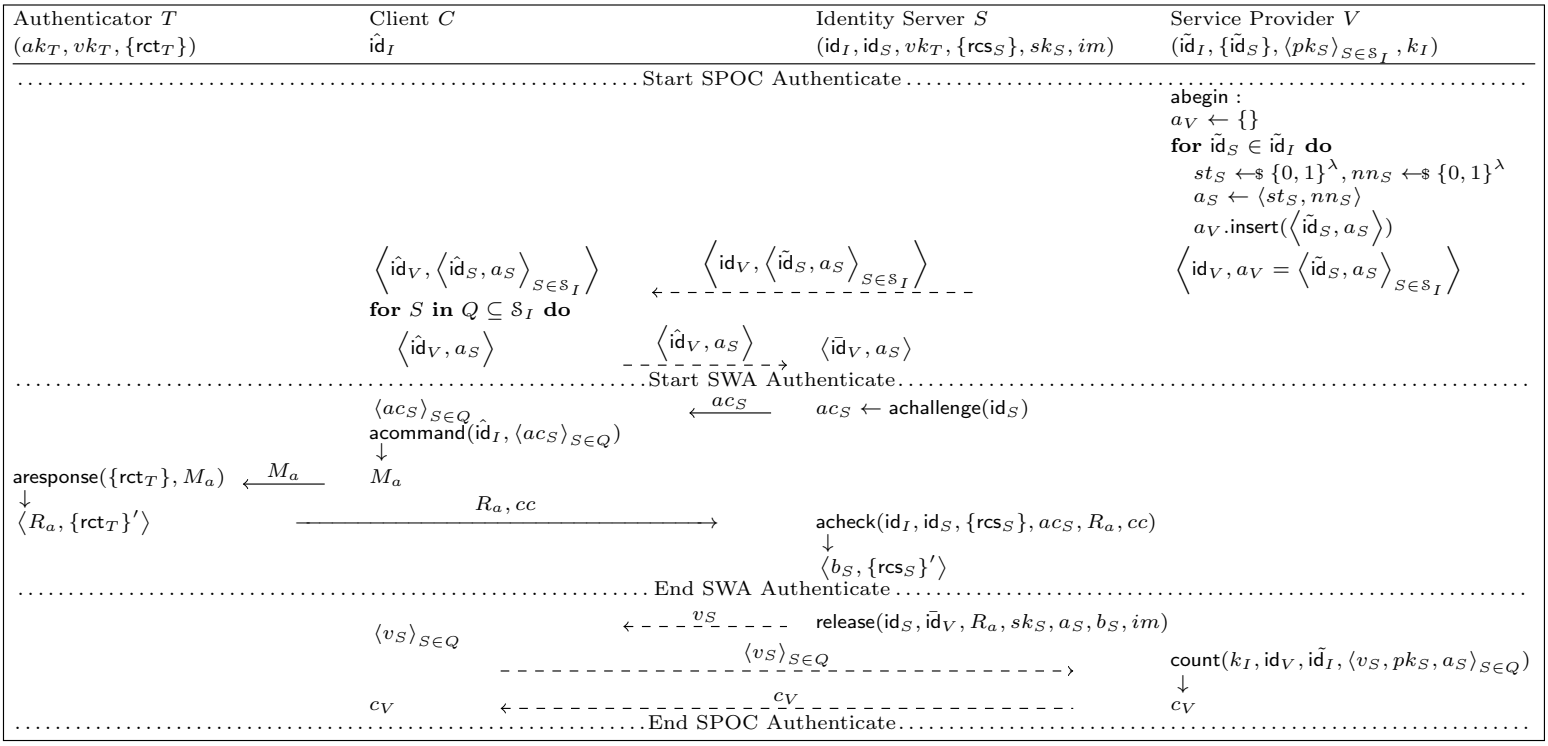

Figure 4.7: Survivable Passwordless OIDC Authenticate and Count subprotocols: solid arrow = authenticated, dashed arrow = authenticated and confidential

Theorem 2 (SPOC security)**.** *If* Sps.Refresh *is a secure registration context refresh protocol, and* $|Q|_{min} = 2k+1$, $n \in [2k+1, 3k+1]$ *then, for any efficient adversary* $\mathcal{A}$ *against the SPOC protocol that makes at most* α *queries to* Init*, there exist efficient adversaries* $\mathcal{D}$ *and* $\mathcal{E}$ *such that:*

$$\mathsf{Adv}^{\mathsf{sps}}_{\mathrm{SPOC}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{spc}}_{\mathrm{SWA}}(\mathcal{E}) + \mathsf{Adv}^{\mathsf{euf-cma}}_{\mathsf{Sig}}(\mathcal{D}) + \alpha^2 \cdot 2^{-\lambda}$$

4.7 Experimental results

We have implemented the authentication subsystem of the architecture (Figure 4.1) that is detailed in Figure 4.9. The main subprotocols are labeled with numbers where (1) is Server Key Generation (and distribution), (2) is Authenticate and (3) is Count. The Web services (identity provider, identity servers and service provider) are implemented through a NGINX 1.18.0 reverse proxy and Flask 2.0.2/GUnicorn 20.1.0 backend. The Web browser is Google Chrome 96, which allows to use as authenticator the Google Virtual WebAuthn Authenticator that mimics in software the behavior of CTAP2 devices. In this way, we

$\mathsf{release}(\mathsf{id}_S, \tilde{\mathsf{id}}_V, R_a, sk_S, a_S, b_S, im)$

1 : **if** $b_S = 0$ **return** $\perp$
2 : $\langle st_S, nn_S \rangle \leftarrow a_S$
3 : $\langle cid, ad, \sigma, auid, cc \rangle \leftarrow R_a$
4 : $\mathsf{sid}_{SPC} \leftarrow \langle ad, \mathsf{H}(cc) \rangle$
5 : $uid \leftarrow im[auid]$
6 : $t \leftarrow \left\langle uid, \mathsf{id}_S, \tilde{\mathsf{id}}_V, \mathsf{sid}_{SPC}, nn_S \right\rangle$
7 : $\sigma \leftarrow \mathsf{Sig.Sign}(sk_S, t)$
8 : **return** $\langle \langle t, \sigma \rangle, st_S \rangle$

(a) Release algorithm

$\mathsf{count}(k_I, \mathsf{id}_V, \tilde{\mathsf{id}}_I, \langle v_S, pk_S, a_S \rangle_{S \in Q})$

1 : **if** $|d| < 2k_I + 1$ **return** reject
2 : $m \leftarrow \mathsf{dict}()$
3 : **for** S **in** Q **do**
4 : **if** $\mathsf{Sig.Verify}(pk_S, t, \sigma) = 0$ **continue**
5 : **if** $v_S.st_S \neq a_S.st_S$ **continue**
6 : **if** $v_S.t.\mathsf{id}_S \neq \mathsf{id}_S$ **continue**
7 : **if** $v_S.t.\mathsf{id}_S \notin \tilde{\mathsf{id}}_I$ **continue**
8 : **if** $v_S.t.\mathsf{id}_V \neq \mathsf{id}_V$ **continue**
9 : **if** $v_S.t.nn_S \neq a_S.nn_S$ **continue**
10 : $j \leftarrow \langle v_S.t.uid, v_S.t.\mathsf{sid}_{SPC} \rangle$
11 : **if** j **not in** m
12 : $m[j] \leftarrow 1$
13 : **else**
14 : $m[j] \leftarrow m[j] + 1$
15 : **if** $m[j] \geq 2k_I + 1$
16 : **return** $c_V \leftarrow\!\!\$\ \{0,1\}^\lambda$
17 : **return** reject

(b) Count algorithm

Figure 4.8: SPOC count and release algorithms

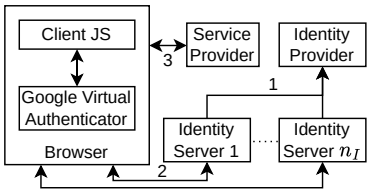

Figure 4.9: Architectural diagram of the prototype

can automate our testbed and exclude non-deterministic user mediation that is, physical interaction with the authenticator. The code running on the Identity Servers extends Python-FIDO2 [120], a library developed by Yubico to implement a WebAuthn relying party. The code running on the browser is a set of custom Javascript scripts that implement the Authenticate procedure with asynchronous and concurrent requests whenever possible.

We propose a performance evaluation of the authentication procedure with the aim of verifying that it meets acceptable requirements in terms of response times. (For this reason, a full OIDC-compliant implementation of the *implicit flow* is out of the scope for this proposal.) We follow the Google RAIL user-centric performance model [62] that breaks down user experience into key actions (tap, scroll, click, load) carried out by the Web browser as a sequence of tasks (loading pages, changing views). This approach helps us to define performance goals for each action. We know that delays above 1 second for an action should be avoided because they induce users to lose their focus on the task they are performing. Thus, our main goal is to assess whether the authentication pro-

cedure started by clicking the "Sign In" interface button terminates within 1 second.

We define the end-to-end authentication time t_{auth} as the time interval occurring between the beginning of (2) (the user clicks on the "Sign In" button) and the end of (3) (the Web browser receives the Count subprotocol response from the Service Provider). We use the Chrome DevTools Performance Tab functions to obtain a detailed profile of the authentication procedure, which is processed with custom Python scripts to compute t_{auth}. We ignore deliberately the registration procedure as it only occurs once per user and has response times compatible with authentication. We do not consider signature computation times of non-virtual authenticators because they are out of the protocol's scope.

4.7.1 Testbed

The prototype components shown in Figure 4.9 are executed in containers deployed in three different scenarios: local, continental, intercontinental.

Local scenario. Every container executes on a single off-the-shelf desktop with the following characteristics: Intel i7 8665U CPU running at 1.9GHz, WDC PC SN730 Sandisk (500GB), 16GB RAM. The average latency between the Web browser and the containers is negligible. This scenario represents the lowest bound of t_{auth} in the most favorable network conditions.

Continental scenario. Each container is executed as a shared `e2-standard-2` Google Compute Engine instance with the following characteristics: 2 virtual Intel Xeon(R) CPUs running at 2.2GHz, Google PersistentDisk (16GB), 8GB RAM. The region is `europe-west` (Belgium, Germany, England). The average ICMP round-trip time to the client is in the order of 35ms. This scenario evaluates t_{auth} in a network setup where all nodes are reasonably close to the client.

Intercontinental scenario. The container characteristics are the same as those in the Continental scenario, but each container executes in a different region among `asia-east`, `europe-west`, `us-central` and `australia-south`. The average ICMP round-trip time to the client is in the order of 250ms. This scenario evaluates t_{auth} in a network where most nodes are reasonably far from the client.

We evaluate t_{auth} also for different combinations of the security threshold k (the amount of tolerated malicious servers). Since the Continental and Intercontinental scenarios are subject to latency fluctuations, we take 30 measurements of t_{auth} and present them as boxplots with averages (dashed line) and medians (solid line).

4.7.2 Evaluation

Figure 4.10 compares t_{auth} for all scenarios in a baseline setup characterized by $k = 0$ and $n_I = 1$. This setup does not tolerate any malicious identity server that is, our protocol is de facto disabled. The goal is to provide a lower bound to t_{auth} that may serve as a reference for subsequent experiments.

The overhead of our implementation can be assessed by analyzing the Local

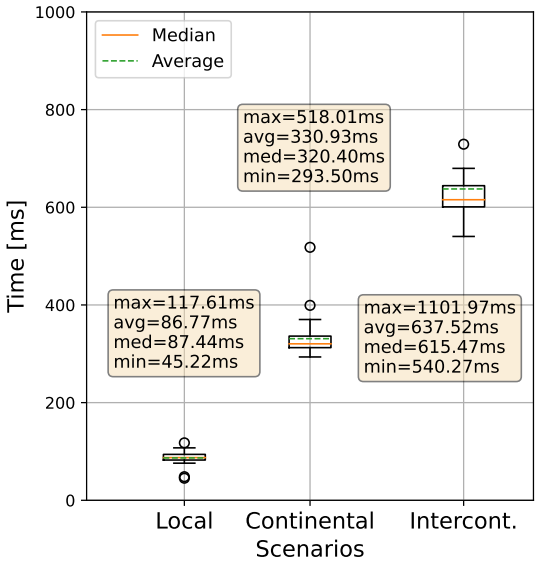

Figure 4.10: Baseline setup comparison in different scenarios

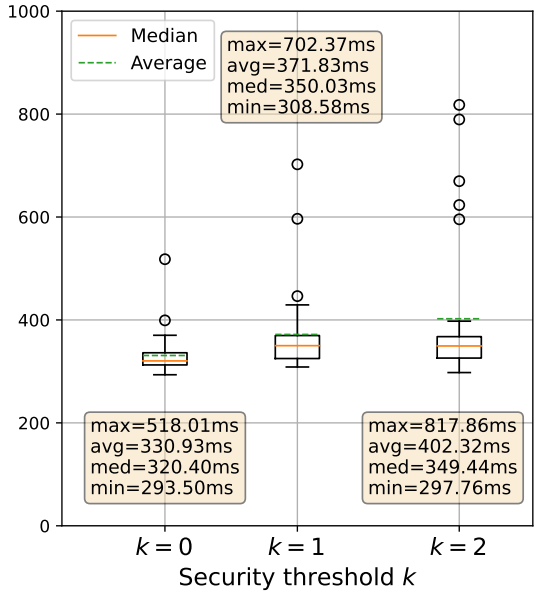

Figure 4.11: Overhead of varying security thresholds

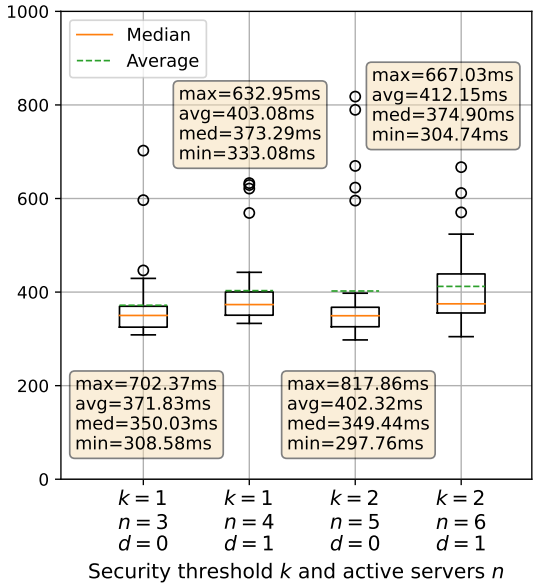

Figure 4.12: Overhead of a failing server

scenario that has negligible network overhead. As expected, the distribution of t_{auth} exhibits low variance with samples ranging from 45.22ms to 117.61ms. In geographically distributed scenarios, we observe that the prototype executes well within the target 1s deadline even in the most challenging Intercontinental scenario. We can observe an interesting trade-off between performance and failure independence: more distant identity servers give higher guarantees of failure independence in terms of availability at the cost of increased network latency. If we make the reasonable assumption that the nodes offered by Google Computing Platform in different regions are failure-indipendent, the Continental scenario offers the best trade-off.

Figure 4.11 compares t_{auth} for varying values of $k = 0, 1, 2$ and $n_I = 2k + 1$ in the Continental scenario. The goal is to assess the impact on t_{auth} of an increased level of tolerance to malicious identity servers.

The protocol overhead is limited. Average t_{auth} increases at most by 21.6% when shifting from $k = 0$ to $k = 2$. If we consider the worst-case (outliers), the maximum t_{auth} increases by circa 50%. The reason behind this behavior lies in the asynchronous prototype implementation that allows concurrent authentication requests to the identity servers. Since the overhead of k on the protocol is low, the choice of k depends on the ability to solve the economic and technological challenges behind the design, implementation and deployment of $2k + 1$ failure-independent nodes at the hardware and software level. An interesting trade-off is between the level of failure-independence (application, libraries, OS or hardware) and the economic investment needed to achieve it. In this work, we

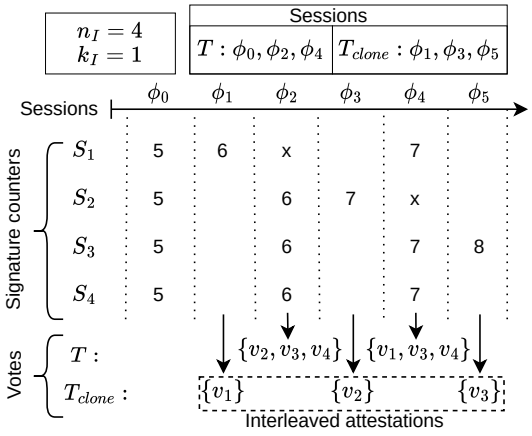

Figure 4.13: Clone detection evasion via authentication interleaving

limit k to 1 or 2, because $k \geq 3$ constitutes a technological challenge with often prohibitive costs. Even current literature on intrusion tolerance (e.g., Byzantine fault-tolerant systems) also assumes $k = 1, 2$ when evaluating performance (e.g., [73, 24, 82]). For the Continental scenario, the most adequate value for k is 1 or at most 2.

Figure 4.12 considers the same prototype in the Continental scenario under two different setups to assess the impact on t_{auth} of a failing identity server. In the first one no identity server may crash ($d = 0$). In the second scenario, one identity server may crash ($d = 1$). We compare the results for both $k = 1$ and $k = 2$.

As one can expect, the boxplots show a slight increase of the authentication time when an identity server fails: for $k = 1$ the average increases is from 371.83ms to 403.08ms (8%) while for $k = 2$ the increment is below 2%. The main reason behind this result is that the prototype code running concurrently on the Web browser sends identity tokens to the service provider as soon as it receives the first $2k + 1$ responses from the identity servers. We can conclude that the protocol incurs in no tangible performance loss when an identity server fails. All these results demonstrate that the proposed protocol is theoretically robust and even usable in real scenarios.

4.8 Attacks examples

4.8.1 Example of clone detection evasion

Figure 4.13 shows an example of clone detection evasion where T and T_{clone} denote the legitimate authenticator and its clone. We denote as $\phi_i, i \in [0, 5]$ the challenge-response sessions participated by authenticators, and as v_j the identity attestation released by identity server j. Denied authentications are denoted as x. Authenticators T and T_{clone} execute the protocol in interleaved sessions: T_{clone} executes sessions ϕ_1, ϕ_3 and ϕ_5, while T executes sessions ϕ_2

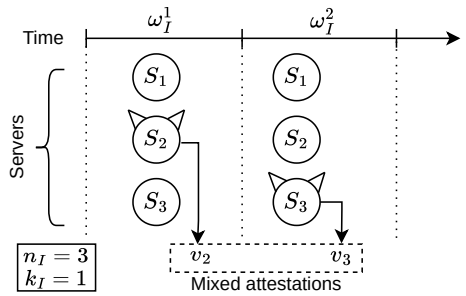

Figure 4.14: Attestation mixing

and ϕ_4. In each session, T_{clone} only collects attestations from single distinct identity servers. All T sessions receive a denied authentication from the server that interacted with T_{clone} because, from the server's view, T presents a stale counter. However, T completes successfully because it receives a number of denied authentications that does not exceed the number of allowed malicious server $k_I = 1$. Thus, T does not detect the presence of T_{clone}. As a result, T_{clone} is able to authenticate after ϕ_5 by presenting interleaved identity attestations $\{v_1, v_2, v_3\}$.

4.8.2 Attestation mixing

Figure 4.14 shows an example of identity attestation mixing in which malicious identity servers S_2 and S_3 release rogue attestations v_2 and v_3 in distinct time periods ω_I^1 and ω_I^2. The attack is successful if the service provider counts identity attestations v_2 and v_3 together and does not detect that they belong to distinct time periods.

4.8.3 Shared session attack

Figure 4.15 shows an example of a shared session attack. The adversary controls identity server S_1 and is correctly registered at all remaining identity servers, which are assumed to be honest. The client starts SPS authentication with service provider V and forwards SPS session information to each identity server, as required by the protocol. The adversary obtains the victim session information and starts SPC authentication with honest identity servers by using the victim session information. As a result, each identity server issues an identity attestation of the adversary identity which is bound to the victim session information. Meanwhile, the client has opened a parallel communication channel with the adversary as a result of other attacks (e.g., CSRF, phishing). To conclude the attack the adversary builds an identity attestation collection and injects it into the parallel channel. The client sends the injected attestation collection to the service provider which authenticates the client under the adversary identity.

4.9 Survivable WebAuthn (SWA) security proof

We extend the proof of [16], which demonstrates the security of non-survivable WebAuthn, to prove the security of the proposed Survivable WebAuthn (SWA)

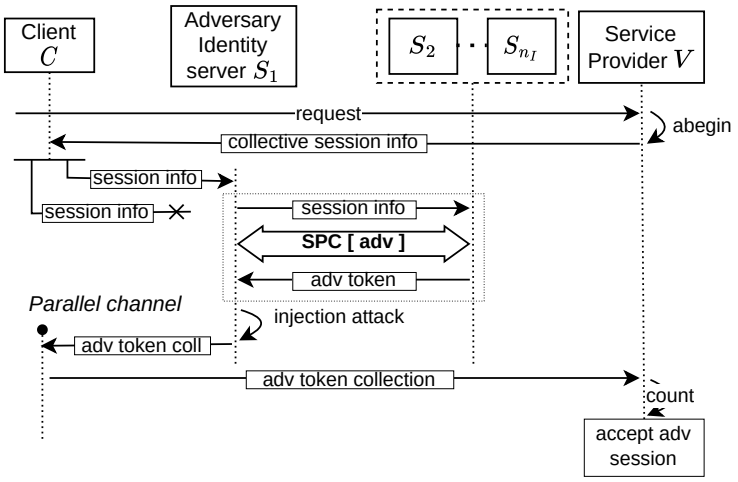

Figure 4.15: Shared session attack

protocol defined in Theorem 1. As in [16] we follow the sequence of games technique of [111].

Proof of Lemma 1. The adversary wins Experiment 2 in three cases: i) if the challenger produces at least a registration context $\mathbf{v}_i = \langle auid, cid, pk, sc, cc, \sigma \rangle$ for an honest identity server $i \notin \Sigma$ such that the new credential $\langle auid, cid, pk \rangle$ is different from the honest server credential $\langle auid_i, cid_i, pk_i \rangle$, ii) the authenticated signature counter value sc is strictly less than the honest server signature counter value sc_ℓ^i, or iii) the authenticated signature counter value sc is greater than the signature value sc_ℓ^i of any honest identity server $i \notin \Sigma$. The first condition cannot hold because the honest majority of identity servers ($|\Sigma| \leq k$ and $n \in [2k+1, 3k+1]$) shares the same credential $\langle auid_i, cid_i, pk_i \rangle$. The second condition cannot hold because the new signature counter value sc is set equal to the maximum authenticated signature counter value of credential $\langle auid, cid, pk \rangle$ among all identity servers, and thus sc is guaranteed to be monotonically increasing on all identity servers. If the third condition holds, we can construct an adversary $\mathcal{D}$ that wins the existential unforgeability game of signature scheme Sig. □

Proof of Theorem 1. Consider a sequence of games and let $\Pr_i$ be the probability of adversary $\mathcal{A}$ winning **Game** i.

Game 0: This game proceeds exactly as in Experiment 1, so $\Pr_0 = \mathsf{Adv}^{\mathrm{spc}}_{\mathrm{SWA}}(\mathcal{A})$.

Game 1: This game proceeds as Game 0, except that it aborts if a hash collision on H occurs. For example, this may allow $\mathcal{A}$ to replay a past response to honest identity servers. If a hash collision occurs, it is easy to construct an efficient adversary $\mathcal{B}$ against the collision-resistance of H (i.e. $\mathcal{B}$ prints the collision). Thus we have that $|\Pr_0 - \Pr_1| \leq \mathsf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{B})$.

Game 2: This game proceeds as Game 1, except that it aborts if there exists a collision on collective challenge cc, which occurs if $\mathsf{belongs}(cc, ch_S, \mathsf{id}_S) = 1 \wedge \mathsf{belongs}(cc, ch'_S, \mathsf{id}_S) = 1$, where $\mathsf{belongs}$ returns 1 if challenge ch_S is associated to identity id_S in data structure cc, 0 otherwise. This would allow $\mathcal{A}$ to replay a past response to honest identity servers. Associative array cc guarantees that each challenge is scoped to a well-defined identity server id_S. Thus, condition $\mathsf{belongs}(cc, ch_S, \mathsf{id}_S) = 1 \wedge \mathsf{belongs}(cc, ch'_S, \mathsf{id}_S) = 1$ holds only if $ch_S = ch'_S$. As a result, the probability that two identical instances of cc occur is at most equal to the probability of an honest identity server producing a pair of identical challenges ch_S. Assuming $\mathcal{A}$ makes at most η queries to Start, the probability that there is at least a pair of equal ch_S challenges among η is, from the union bound, less or equal to $\binom{\eta}{2} \cdot 2^{-\lambda} \leq \eta^2 \cdot 2^{-\lambda}$.

Game 3: This game aborts if an honest server oracle outputs an accept vote without having a unique authenticator oracle partner. If such an event occurs, we can construct an efficient adversary $\mathcal{D}$ against existential unforgeability of signature scheme Sig. Adversary $\mathcal{D}$ guesses the offending server oracle $\pi_S^{i,j}$ with a probability of at least $1/\theta$, and then simulates the game by answering all queries related to the credential involved in session $\pi_S^{i,j}$ with the signing oracle $\mathsf{Sig.Sign}$. If the server oracle $\pi_S^{i,j}$ accepts without a unique partner then there are two cases: $\pi_S^{i,j}$ has at least two partners, or it has no partner.

We first consider the Registration phase ($j = 0$). If an honest server oracle has at least two partners, it means that at least two authenticator oracles signed the same message. This only happens if at least two authenticator oracles generate the same random cid, which can happen with probability at most $\theta^2 \cdot 2^{-\lambda}$. In fact we do not require that the probabilistic $\mathsf{Sig.Gen}$ algorithm accepts freshly generated randomness. Such randomness can be deterministically derived from the authenticator internal state, and thus cloned authenticators may generate the same credentials. Instead, if an honest server oracle has no partners, then $\mathcal{D}$ has forged a valid signature and wins the existential unforgeability game.

We now consider the Authentication phase ($j \geq 1$). In Authentication we know that if server oracle $\pi_S^{i,j}$ outputs an accept vote, then the corresponding server registration oracle $\pi_S^{i,0}$ verified a signature with public key pk established during registration with its unique partner authenticator oracle $\pi_T^{q,0}$. So if $\pi_S^{i,j}$ has at least two partners, it means that they are either sessions of the same authenticator oracle following $\pi_T^{q,0}$ or sessions of authenticators cloned after $\pi_T^{q,0}$ completed registration and thus sharing the same credentials in their registration contexts. However, this is not possible because an authenticator oracle and honest identity server oracles increment the signature counter sc in every authentication session. Moreover, for Lemma 1 honest identity servers are guaranteed to have a monotonically increasing value of sc after each Refresh phase. Instead, if $\pi_S^{i,j}$ has no partner but still accepts despite $\pi_S^{i,0}$ being partnered with $\pi_T^{q,0}$, then this means that $\mathcal{D}$ has forged a valid signature and thus wins the existential unforgeability game. As a result we have that $|\mathrm{Pr}_2 - \mathrm{Pr}_3| \leq \theta \cdot \mathsf{Adv}_{\mathsf{Sig}}^{\mathsf{euf-cma}}(\mathcal{D}) + \theta^2 \cdot 2^{-\lambda}$.

Game 4: This game aborts if there exist disjoint subsets of $|Q|_{min} - k$ honest

identity server oracles that output accept and are partnered with the same authenticator oracle. This cannot happen if $n - k - (|Q|_{min} - k) < |Q|_{min} - k$. The configuration $|Q|_{min} = 2k + 1$ and $n \in [2k + 1, 3k + 1]$ satisfies the said condition.

Final analysis: The adversary now has probability 0 of winning the experiment as unique partnership is guaranteed, session identifiers $\langle ad, \mathsf{H}(cc)\rangle$ cannot collide, and no disjoint groups of honest identity servers output accept for the same authenticator oracle. □

4.10 Survivable Passwordless OpenID Connect (SPOC) security proof

According to Definition 4.6.4 the adversary wins Experiment 3 if it can break secure authentication (Definition 4.6.1) and session integrity for authentication (Definition 4.6.2). The adversary breaks secure authentication if a fresh service provider oracle π_V^h accepts and any of the following conditions does *not* hold:

1. π_V^h is partnered with a set of at least $|Q|_{min}$ identity server oracles $\{\pi_S^{i,j}\}$;
2. each oracle $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ is an oracle of a distinct identity server S;
3. each oracle $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ is an oracle of an identity server S that belongs to the same identity provider I;
4. each oracle $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ is uniquely partnered with an authenticator oracle $\pi_T^{q,l}$;
5. there do not exist distinct disjoint subsets of $|Q|_{min} - k_I$ identity server oracles $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_I}$ that output accept and are partnered with $\pi_T^{q,l}$;
6. all oracles $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ are partnered with the same authenticator oracle $\pi_T^{q,l}$;
7. π_V^h is uniquely partnered with π_C^g;
8. the adversary cannot win the indistinguishability game.

Moreover, it breaks session integrity for authentication if π_V^h outputs accept and any of the following conditions does *not* hold:

9. for π_V^h, $st \neq \epsilon$ and $nn \neq \epsilon$, where ϵ denotes the empty string;
10. π_V^h has output accept for identity *uid* and is partnered with a unique set of at least $|Q|_{min}$ identity server oracles $\{\pi_S^{i,j}\}$ that have output accept on the same identity *uid*.

Proof sketch of Theorem 2. Let's consider the requirements listed above.

(1) The count algorithm accepts only if π_V^h has received $|Q|_{min} = 2k_I + 1$ identity attestations. Requirement 1 does not hold if there is a collision on any pair $\langle st_S, nn_S \rangle$, which can happen with probability at most $\alpha^2 \cdot 2^{-\lambda}$, where α is the number of Init queries allowed to $\mathcal{A}$. Otherwise, given that the client is honest and that pairs $\langle st_S, nn_S \rangle$ are sent over confidential and authenticated communication channels, then π_V^h is guaranteed to be partnered with $|Q|_{min}$ identity server oracles;

(2, 3) If π_V^h is partnered with at least two oracles $\pi_S^{i,j}$ of the same identity server S, or is partnered with at least two oracles $\pi_S^{i,j}$, $\pi_W^{k,r}$ such that $\mathsf{id}_S \in \hat{\mathsf{id}}_I \wedge \mathsf{id}_W \in \bar{\mathsf{id}}_I$, and $\hat{\mathsf{id}}_I \neq \bar{\mathsf{id}}_I$, then we can construct an adversary $\mathcal{D}$ against the existential unforgeability of signature scheme Sig used to authenticate identity attestations in the Sps.release algorithm;

(4, 5) We can prove that 4 and 5 hold by a direct reduction to the SPC game: we can construct an adversary $\mathcal{E}$ that simulates the SPS experiment and uses the Start, Challenge and Complete queries of the SPC experiment to simulate the Start and Complete queries of the SPS experiment. Given that there is no restriction on adversarial queries in the SPC experiment, if 4 or 5 are violated in the SPS experiment, then they are also violated in the SPC experiment;

(6) The count algorithm accepts only if all identity attestations include the same sid_{SPC}. For the unique partnership property proved above, it follows that all identity server oracles are partnered to the same authenticator oracle;

(7) Requirement 7 does not hold if π_V^h accepts but its partnership is not unique. If π_V^h is partnered with zero client oracles, then we can construct and adversary $\mathcal{D}$ against existential unforgeability of the signature scheme Sig used to authenticate identity attestations in the Sps.release algorithm. If π_V^h is partnered with at least two client oracles, then two client oracles share the global session identifier. However, session cookie c_V is freshly chosen at each Sps.count execution and thus global session identifiers can collide only with negligible probability.

(8) If $\mathcal{A}$ wins the indistinguishability game then we can construct an efficient adversary $\mathcal{E}$ that can read confidential channels, thus contradicting our assumptions on confidential channels.

(9) The count algorithm outputs accept only if values nn and st of π_V^h are not equal to ϵ, which can only happen if π_V^h has executed abegin.

(10) Service provider oracle π_V^h outputs accept on identity uid only if it has received at least $|Q|$ attestations on identity uid. Thus, if $uid \neq uid'$, where uid' is the user identity voted by honest identity server oracles in Q, then the adversary is able to send $|Q| - k$ attestations $\{v_S\}$ authenticated by honest identity servers $\mathsf{id}_S \notin \Sigma^i$ such that $v_S.t.st_S = a_S.st_S \wedge v_S.t.nn_S =$

$a_S.nn_S$. However, this is not possible because the adversary does not know values nn_S and st_S of honest identity servers $\mathsf{id}_S \notin \Sigma^i$.

□

4.11 Composition of SPOC and CTAP2

In this section we demonstrate that the composition of SPOC with the CTAP2 protocol of the FIDO2 specification is secure. In Sections 4.11.1 and 4.11.2 we give the due background knowledge on CTAP2. In Section 4.11.1 we report the CTAP2 protocol specification by referring to the *PIN-based access control for authenticators* (PACA) operations framework introduced in [16]. In Section 4.11.2 we focus on PACA security definitions that apply to CTAP2 and omit stronger security definitions that do not capture the security of any currently standardized CTAP2 version. In Section 4.11.3 we give the composed protocol operations framework and in Section 4.11.4 its security model. Finally, in Section 4.11.5 we prove the security of the composed protocol.

4.11.1 PIN-based Access Control for Authenticators (PACA) operations framework

We first give a minimal description of how [16] models user interaction, and refer the interested reader to the original paper for further details. User interaction is modeled as a predicate $\{0,1\} \leftarrow \mathsf{G}(x,y)$ on information x input to the client, and information y input to the authenticator. This allows to capture a user playing the role of an out-of-band secure channel that validates the consistency of information exchanged between the authenticator and the client.

We now give the verbatim PACA specification as introduced in [16]. A PACA protocol is an interactive protocol involving a human user, an authenticator, and a client. The state of authenticator T, denoted by st_T, consists of static storage $\mathsf{st}_T.\mathsf{ss}$ that remains intact across reboots and volatile storage $\mathsf{st}_T.\mathsf{vs}$ that gets reset after each reboot. $\mathsf{st}_T.\mathsf{ss}$ is comprised of: i) a private secret $\mathsf{st}_T.\mathsf{s}$ and ii) a public retries counter $\mathsf{st}_T.\mathsf{n}$, where the latter is used to limit the maximum number of consecutive failed active attacks (e.g., PIN guessing attempts) against the authenticator. $\mathsf{st}_T.\mathsf{vs}$ consists of: i) power-up state $\mathsf{st}_T.\mathsf{ps}$ and ii) binding states $\mathsf{st}_T.\mathsf{bs}_i$ (together denoted by $\mathsf{st}_T.\mathsf{bs}$). A client C may also keep binding states, denoted by $\mathsf{bs}_{C,j}$. All states are initialized to the empty string ϵ.

A PACA protocol is associated with an arbitrary public gesture predicate G and consists of the following algorithms and subprotocols, all of which can be executed a number of times, except if stated otherwise:

- **Reboot**: this algorithm represents a power-down/power-up cycle and it is executed by the authenticator with mandatory user interaction. We use $\mathsf{st}_T.\mathsf{vs} \leftarrow\!\!\$\ \mathsf{reboot}(\mathsf{st}_T.\mathsf{ss})$ to denote the execution of this algorithm, which inputs its static storage and resets all volatile storage. Note that one should always run this algorithm to power up the aythenticator at the beginning of PACA execution.

- **Setup**: this subprotocol is executed at most once for each authenticator. The user inputs a PIN through the client and the authenticator inputs its volatile storage. In the end, the authenticator sets up its static storage and the client (and through it the user) gets an indication of whether the subprotocol completed successfully.

- **Bind**: this subprotocol is executed by the three parties to establish an access channel over which commands can be issued. The user inputs its PIN through the client, whereas the authenticator inputs its static storage and power-up state. At the end of the subprotocol, each authenticator and client terminating successfully get a (volatile) binding state and sets the session identifier. In either case (success or not), the authenticator may update its static retries counter. We assume the client always initiates this subprotocol once it gets the PIN from the user.

- **Authorize**: this algorithm allows a client to generate authorized commands for the authenticator. The client inputs binding state $\mathsf{bs}_{C,j}$ and command M. We denote $\langle M, t\rangle \leftarrow\!\!\$\ \mathsf{authorize}(\mathsf{bs}_{C,j}, M)$ as the generation of an authorized command.

- **Validate**: this algorithm allows a authenticator to verify authorized commands sent by a client with respect to a user decision (where the human user inputs the public gesture predicate G). The authenticator inputs a binding state $\mathsf{st}_T.\mathsf{bs}_i$, an authorized command $\langle M, t\rangle$, and a user decision $d = \mathsf{G}(x, y)$. We denote $b \leftarrow \mathsf{validate}(\mathsf{st}_T.\mathsf{bs}_i, \langle M, t\rangle, d)$ as the validation performed by the authenticator to obtain an accept or reject indication.

Correctness (PACA). For an arbitrary public predicate G, we consider any authenticator T and any sequence of PACA subprotocol executions that includes the following (which may not be consecutive): i) a Reboot of T; ii) a successful Setup using PIN fixing $\mathsf{st}_T.\mathsf{ss}$ via some client; iii) a Bind with PIN creating authenticator-side binding state $\mathsf{st}_T.\mathsf{bs}_i$ and client-side binding state $\mathsf{bs}_{C,j}$ at a client C; iv) authorization of command M by C as $\langle M, t\rangle \leftarrow\!\!\$\ \mathsf{authorize}(\mathsf{bs}_{C,j}, M)$; and v) validation by T as $b \leftarrow \mathsf{validate}(\mathsf{st}_T.\mathsf{bs}_i, \langle M, t\rangle, d)$. If no Reboot of T is executed after iii), then correctness requires that $b = 1$ if and only if $\mathsf{G}(x, y) = 1$ (i.e., $d = 1$) holds.

4.11.2 PACA security model

Trust assumptions. As we are interested in capturing CTAP2 security guarantees, we directly inherit the trust assumptions required to prove CTAP2 secure. In particular, we assume that Setup is executed on an authenticated, yet non-confidential, communication channel between the client and the authenticator. Moreover, we assume that the client is honest, that is, the adversary cannot corrupt any client instance that is bound to the authenticator, nor can it mount active attacks against clients during the Binding execution. However, the adversary can still launch active attacks against authenticators.

Session oracles. To capture parallel instances of PACA protocols, each party $P \in \mathcal{C} \cup \mathcal{T}$ is associated with a set of oracles $\{\pi_P^i\}$, where π_P^i models the i-th instance of P.

Partnership. An authenticator oracle π_T^i and a client oracle π_C^j are each other's partner if both have completed their binding executions and share the same session identifier, which must be specified by the PACA protocol. When an authenticator is rebooted then all of its existing session oracles are invalidated.

We now give the PACA security experiment, which extends the PACA security game defined in [16] by introducing Clone queries.

Security experiment 4 (PACA)**.** *The security experiment is executed between a challenger and an adversary $\mathcal{A}$. At the beginning of the experiment, the challenger fixes an arbitrary distribution D over PIN dictionary $\mathcal{PIN}$ associated with PACA; it then samples independent user PINs according to D, denoted by $\left\langle \mathsf{pin}_U \xleftarrow{D} \mathcal{PIN} \right\rangle_{U \in \mathcal{U}}$. Without loss of generality, we assume that each user holds only one PIN. The challenger also initializes states of all oracles to the empty string. Then, $\mathcal{A}$ is allowed to interact with the challenger via the following queries:*

- $\mathsf{Reboot}(T)$*: the challenger runs Reboot for authenticator T, marking all previously used instances π_T^i (if any) as* invalid *and setting* $\mathsf{st}_T.\mathsf{vs} \leftarrow\!\!\$\ \mathsf{reboot}(\mathsf{st}_T.\mathsf{ss})$.

- $\mathsf{Setup}(\pi_T^i, \pi_C^j, U)$*: the challenger inputs pin_U through π_C^j, runs Setup between π_T^i and π_C^j; it returns the trace of communications to $\mathcal{A}$. After this query, T is* set up*, i.e., $\mathsf{st}_T.\mathsf{ss}$ is set and available for the rest of the experiment. Oracles created in this query, i.e. π_T^i and π_C^j, must never have been used before and are always marked* invalid *after Setup completion.*

- $\mathsf{Execute}(\pi_T^i, \pi_C^j)$*: the challenger runs Bind between π_T^i and π_C^j using the same pin_U that set up T; it returns the trace of communications to $\mathcal{A}$. This query allows the adversary to access honest Bind executions in which it can only take passive actions, i.e., eavesdropping. The resulting binding states on both sides are kept as $\mathsf{st}_S.\mathsf{bs}_i$ and $\mathsf{bc}_{C,j}$ respectively.*

- $\mathsf{Connect}(T, \pi_C^j)$*: the challenger asks π_C^j to initiate the Bind subprotocol with T using the same pin_U that set up T; it returns the first message sent by π_C^j to $\mathcal{A}$. Note that no client oracles can be created for active attacks if* $\mathsf{Connect}$ *queries are disallowed, since we assume the client is the initiator of Bind. This query allows the adversary to launch an active attack against a client oracle.*

- $\mathsf{Send}(\pi_P^i, m)$*: the challenger delivers m to π_P^i and returns its response (if any) to $\mathcal{A}$. If π_P^i completes the Bind subprotocol, then the binding state is kept as $\mathsf{st}_T.\mathsf{bs}_i$ for an authenticator oracle and as $\mathsf{bs}_{C,i}$ for a client oracle. This query allows the adversary to launch an active attack against an authenticator oracle or completing an active attack against a client oracle.*

- $\mathsf{Clone}(T)$*: the challenger marks* T *as* cloned*, adds a new authenticator* T' *to* $\mathcal{T}$*, marks* T' *as* cloned*, sets* T' *static storage equal to* T *static storage (*$\mathsf{st}_T.\mathsf{ss}$*), and returns the static storage to* $\mathcal{A}$*.*
- $\mathsf{Authorize}(\pi_C^j, M)$*: the challenger asks* π_C^j *to authorize command* M*; it returns the authorized command* $\langle M, t\rangle \leftarrow\!\$\ \mathsf{authorize}(\mathsf{bs}_{C,j}, M)$*.*
- $\mathsf{Validate}(\pi_T^i, \langle M, t\rangle)$*: the challenger asks* π_T^i *(that received a user decision* d*) to validate* $\langle M, t\rangle$*; it returns the validation result* $b \leftarrow \mathsf{validate}(\mathsf{st}_T.\mathsf{bs}_i, \langle M, t\rangle, d)$*.*
- $\mathsf{Compromise}(\pi_C^j)$*: the challenger returns* $\mathsf{bs}_{C,j}$ *and marks* π_C^j *as* compromised*.*
- $\mathsf{Corrupt}(U)$*: the challenger returns* pin_U *and marks* pin_U *as* corrupted*.*

Security goals. We focus on the *unforgeability with trusted binding* (UF-t) advantage measure for a PACA protocol Π introduced in [16], which we extend to account for Clone queries. For clarity, we denote our UF-t security formulation as *UF-t-c*.

Definition 4.11.1 (UF-t-c advantage)**.** $\mathsf{Adv}_{\Pi}^{\mathsf{uf-t-c}}(\mathcal{A})$ *is the probability that if there exists an authenticator oracle* π_T^i *that accepts an authorized command* $\langle M, t\rangle$ *for gesture* G *and:*

- T *is not cloned;*
- $\mathcal{A}$ *does not make* Connect *queries;*
- $\mathcal{A}$ *does not corrupt* pin_U *used to setup* T *before* π_T^i *accepted* $\langle M, t\rangle$*;*
- $\mathcal{A}$ *does not make* Compromise *queries on any of* T *partners created after* T*'s last reboot and before* π_T^i *accepted* $\langle M, t\rangle$

then at least one of the following conditions does not *hold:*

1. G *approves* M*, i.e.,* $\mathsf{G}(x, y) = 1$
2. $\langle M, t\rangle$ *was output by one of* T*'s valid partners* π_C^j

We note that for UF-t-c security we do not allow the adversary to win the experiment with a cloned authenticator. We also note that the Clone query reveals static storage $\mathsf{st}_T.\mathsf{st}$ to the adversary which, considering the CTAP2 specification, consists of the user pin digest $\mathsf{st}_T.\mathsf{s} \leftarrow \mathsf{H}(\mathsf{pin}_U)$ and the public signature counter $\mathsf{st}_T.\mathsf{n}$. Considering the CTAP2 specification, the adversary can already obtain user pin digest $\mathsf{st}_T.\mathsf{s}$ via the Connect query with a client C that has already executed the Setup procedure with authenticator T. As a result, the Clone query does not give the adversary any additional information that could not already be obtained via other queries. Therefore, UF-t-c security of CTAP2 reduces to UF-t security of CTAP2, and the authors in [16] already prove that CTAP2 protocol is UF-t secure.

4.11.3 Operations framework of the SPS + PACA composition

The composition of SPS and PACA is an interactive protocol among a human user, authenticator T, client C, a service provider V and a set of identity servers $\mathcal{S}_I$ of identity provider I. The state of authenticator T, denoted by st_T, consists of PACA volatile storage $\mathsf{st}_T.\mathsf{vs}$ and the following static storage: i) a SPS attestation key pair $\langle ak_T, vk_T \rangle$, ii) a set of SPS registration contexts $\mathsf{st}_T.\mathsf{rct}$, and iii) PACA static storage $\mathsf{ss}_T.\mathsf{ss}$. Identity server S keeps registration contexts $\mathsf{st}_S.\mathsf{rcs}$. A client C may keep binding states $\mathsf{bs}_{C,j}$. All states are initialized to the empty string ϵ.

The operations framework is associated with a public gesture predicate G and consists of the following algorithms and subprotocols:

- **Authenticator Key Generation**: same as SPS.
- **IdP Parameters Setup**: same as SPS.
- **Server Key Generation**: same as SPS.
- **Refresh**: same as SPS.
- **Reboot**: same as PACA and should be executed to power up the authenticator before executing the following subprotocols.
- **Setup**: same as PACA.
- **Bind**: same as PACA.
- **Register**: is executed among a human user, authenticator T, a client C and all n_I identity servers $S \in \mathcal{S}_I$ of identity provider I. The user inputs the public gesture predicate G, authenticator T inputs its attestation secret key ak_T and a binding state $\mathsf{st}_T.\mathsf{bs}_i$, client C inputs an intended identity provider identity $\bar{\mathsf{id}}_I$ and a binding state $\mathsf{bs}_{C,j}$, each identity server S inputs its identity id_S and the authenticator attestation public key vk_T. At the end of the execution, if successful, authenticators and identity servers that successfully terminate obtain a new registration context, which may be different, bound to the corresponding user identity *uid*. Note that when authenticator T successfully completes the subprotocol, a server may fail to do so in the same run. Moreover, all parties that terminate successfully obtain a *session identifier*.
- **Authenticate**: is executed among a human user, authenticator T, a client C, a subset Q of identity servers $\mathcal{S}_I$ of identity provider I, and service provider V. The user inputs the public gesture predicate G, the authenticator inputs its registration contexts $\mathsf{st}_T.\mathsf{rct}$ and a binding state $\mathsf{st}_T.\mathsf{bs}_i$, the service provider inputs the intended identities of the identity provider $\tilde{\mathsf{id}}_I$ and its identity servers $\tilde{\mathsf{id}}_S$, the client inputs the intended identity of the identity provider $\hat{\mathsf{id}}_S$ and a binding state $\mathsf{bs}_{C,j}$, each server S in subset $Q \subseteq \mathcal{S}_I$ inputs its own identity $\mathsf{id}_S \in \mathsf{id}_I$ and registration contexts $\{\mathsf{st}_S.\mathsf{rcs}\}$. Each identity server that successfully terminates outputs an

identity attestation v_S, and authenticators and identity servers that successfully terminate may update their registration contexts. At the end of the subprotocol if service provider V successfully validates enough identity attestations, then V generates and sends session cookie c_V to the client, reject otherwise.

Correctness (SPS + PACA). Correctness follows naturally from an extension of the SPS correctness that includes a correct setup of PACA binding states. We omit a formal definition.

4.11.4 SPS + PACA security model

Trust assumptions. As in PACA, we assume that Setup is executed on an authenticated, yet non-confidential, communication channel between the client and the authenticator. We assume that the client is honest and that the adversary cannot mount active attacks against clients during the Binding execution. The adversary can still mount active attacks against authenticators, which are assumed to be tamper-proof (i.e., read only access). As in SPS, we require server-to-client authenticated communication channels and we assume a public key infrastructure (PKI) that binds each public key to its owner's identity id. Contrary to SPS, we drop the assumption of authenticator-to-client mutually authenticated communication channels.

Session oracles. The protocol is executed by session oracles of party $P \in \mathcal{T} \cup \mathcal{S} \cup \mathcal{V} \cup \mathcal{C}$. Authenticators have PACA oracles π_T^i, and registration and authentication SPS oracles $\pi_T^{i,0}$ and $\pi_T^{i,j}$ for $j > 0$. Clients have PACA oracles π_C^j and SPS oracles $\hat{\pi}_C^j$. Identity servers have SPS registration and authentication oracles $\pi_S^{i,0}$ and $\pi_S^{i,j}$ for $j > 0$. Finally, service providers have SPS oracles π_V^i.

Partnership. We do not introduce new notions of partnership. The composed protocol naturally inherits partnership definitions from PACA and SPS protocols.

Security experiment 5 (SPS + PACA)**.** *The challenger executes experiment setup as in SPS Experiment 3 and PACA Experiment 4. The experiment proceeds in rounds as SPS Experiment 3. During each round the challenger accepts the same queries defined in PACA Experiment 4, except for* Clone, Authorize *and* Validate, *that are replaced with the following queries:*

- $\mathsf{Init}(\pi_V^h, \hat{\pi}_C^g, \mathsf{id})$*: the challenger instructs service provider oracle* π_V^i *to execute* $\mathsf{Sps.abegin}(\mathsf{id})$ *and returns all* a_S *values such that* $S \in \Sigma^\ell$ *to* $\mathcal{A}$ *and all* a_S *values to* $\hat{\pi}_C^g$*.*

- $\mathsf{Start}(\pi_C^g, \{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}, \langle c \rangle)$*: the challenger instructs up to* $n_{I_t} - k_{I_t}$ *identity server oracles* $\{\pi_S^{i,j}\}$ *of distinct servers* $S \in \mathcal{S}_{I_t}$ *to execute* $\mathsf{Spc.rchallenge}(\mathsf{id}_S)$ *if* $j = 0$ *or* $\mathsf{Spc.achallenge}(\mathsf{id}_S)$ *if* $j > 0$*. The generated challenges (*$\langle rc \rangle$ *if* $j = 0$, $\langle ac \rangle$ *if* $j > 0$*) are returned to* $\mathcal{A}$*. W.l.o.g. either* $j = 0$ *or* $j > 0$ *for all* $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}$*. The challenger then gives the generated challenges along with challenges* $\langle c \rangle$ *to* π_C^g*, which takes its binding state* $\mathsf{bs}_{C,g}$ *and the identities* id_S *of server oracles* $\{\pi_S^{i,j}\}_{S \in \mathcal{S}_{I_t}}$ *and corresponding identity provider*

id_I *to authorize the command output by* $\mathsf{Spc.rcommand}(\mathsf{id}_I, \langle rc \rangle)$ *if* $j = 0$ *or* $\mathsf{Spc.acommand}(\mathsf{id}_I, \langle ac \rangle)$ *if* $j > 0$ *and returns the resulting command to* $\mathcal{A}$*.*

- $\mathsf{Challenge}(\pi_T^i, \pi_T^{j,h}, \langle M, t \rangle)$*: the challenger delivers an authorized command* $\langle M, t \rangle$ *to* π_T^i*, which executes* $b \leftarrow \mathsf{Paca.validate}(\mathsf{st}_T.\mathsf{bs}_i, \langle M, t \rangle, d)$ *with its binding state* $\mathsf{st}_T.\mathsf{bs}_i$ *and user decision* d *sent to* π_T^i *to validate* $\langle M, t \rangle$*. If validation is successful, the challenger instructs* $\pi_T^{j,h}$ *to process command* M *using* $\mathsf{Spc.rresponse}$ *if* $h = 0$ *or* $\mathsf{Spc.aresponse}$ *if* $h > 0$ *and returns the response to* $\mathcal{A}$*.*

- $\mathsf{Clone}(T)$*: the challenger marks* T *as* cloned*, adds a new authenticator* T' *to* $\mathcal{T}$*, marks* T' *as* cloned*, sets* T' *internal state equal to* T *internal state* $(\langle ak_T, vk_T \rangle, \{\mathsf{rct}_T\}, \mathsf{st}_T.\mathsf{st})$*, and returns the internal state to* $\mathcal{A}$*.*

- $\mathsf{Complete}(\pi_S^{i,j}, \pi_C^g, \mathsf{id}_V, T, R, a_S, cc, auid)$*: if* T *is cloned then the challenger extracts all challenges* c_S *in* cc *and then executes the following: if* $j = 0$ *it computes* $\langle M_r, cc', auid' \rangle \leftarrow \mathsf{Spc.rcommand}(\mathsf{id}_I, \langle c_S \rangle)$*, instructs oracle* $\pi_{T'}^{i,j}$ *for all clones* T' *of* T *to execute* $\langle R_r, \{\mathsf{rct}_{T'}\} \rangle \leftarrow \mathsf{Spc.rresponse}(ak_{T'}, \{\mathsf{rct}_{T'}\}, M_r)$ *and instructs oracle* $\pi_S^{i,j}$ *to execute* $\mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_{T'}, c_S, R_r, cc, auid)$*; if* $j > 0$ *it computes* $\langle M_a, cc \rangle \leftarrow \mathsf{Spc.acommand}(\mathsf{id}_I, \langle c_S \rangle)$*, instructs oracle* $\pi_{T'}^{i,j}$ *for all clones* T' *of* T *to execute* $\mathsf{Spc.aresponse}(\{\mathsf{rct}_{T'}\}, M_a)$ *and instructs oracle* $\pi_S^{i,j}$ *to execute* $\mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R, cc)$*. Then, in any case (*$T$ *cloned or not), the challenger instructs identity server oracle* $\pi_S^{i,j}$ *to execute* $\mathsf{Spc.rcheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, vk_T, rc_S, R, cc, auid)$ *if* $j = 0$ *or* $\mathsf{Spc.acheck}(\mathsf{id}_I, \mathsf{id}_S, \{\mathsf{rcs}_S\}, ac_S, R, cc)$ *if* $j > 0$*. The resulting bit* b_S *is returned to* $\mathcal{A}$*. If* $j > 0$*, then the challenger also instructs server oracle* $\pi_S^{i,j}$ *to execute* $\mathsf{Sps.release}(\mathsf{id}_S, \mathsf{id}_V, R, sk_S, a_S, b_S, im)$ *with subsession information* a_S *and returns the resulting identity attestation to* π_C^g*.*

- $\mathsf{Count}(\pi_V^h, \pi_C^g, \langle v \rangle, \mathsf{id}, a)$*: the challenger instructs service provider oracle* π_V^h *to execute* $\mathsf{Sps.count}$ *with authentication session information* a *on the tuple of identity attestations* $\langle v \rangle$ *and the set of attestations owned by* π_C^g*, if any.* π_V^h *verifies the received attestations with public keys of identity servers owned by identity provider* id*. The challenger returns values* $\langle v \rangle$*,* a*, and the resulting session cookie* c_V *to* π_C^g*.*

- $\mathsf{Reveal}(\pi_P^i)$*: the challenger instructs* π_P^i *(*$P \in \mathcal{C} \cup \mathcal{V}$*) to return session cookie* c_V*, authentication session information* a *and attestations* $\langle v \rangle$*. Oracle* π_P^i*, its partner (if any), and values* c_V*,* a *and* $\langle v \rangle$ *are said* unfresh*.*

- $\mathsf{Test}(\pi_P^i)$*: if* π_P^i *(*$P \in \mathcal{C} \cup \mathcal{V}$*) is fresh, has accepted and owns session cookie* c_V*, then the challenger flips a fair coin* $b \leftarrow\!\!\$\ \{0, 1\}$*. If* $b = 0$ *the challenger returns a random sample from* D_λ*, otherwise it instructs* π_P^i *to return* c_V*. The adversary then outputs bit* b' *and wins the indistinguishability game if* $b' = b$*.* $\mathcal{A}$ *can execute this query only once.*

Let $\langle rc \rangle$ and $\langle ac \rangle$ be registration and authentication challenges, a be session information, $\langle M_r, t_r \rangle$ and $\langle M_a, t_a \rangle$ be authorized commands, G_r and G_a be gestures, and the following be oracles for SPS: π_V^h, $\{\pi_S^{i,j}\}$, $\pi_T^{q,l}$, $\hat{\pi}_C^x$, $\hat{\pi}_{C'}^w$; and PACA: π_T^m, π_T^p, π_C^y and $\pi_{C'}^z$. The adversary wins the game if a fresh identity provider oracle π_V^h outputs accept *and any of the following conditions does not hold:*

1. *π_V^h is partnered with a set of at least $|Q|_{min}$ identity server oracles $\{\pi_S^{i,j}\}$;*
2. *each oracle $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ is an oracle of a distinct identity server S;*
3. *each oracle $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ is an oracle of an identity server S that belongs to the same identity provider I;*
4. *each oracle $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ is uniquely partnered with an authenticator oracle $\pi_T^{q,l}$;*
5. *there do not exist distinct disjoint subsets of $|Q|_{min} - k$ identity server oracles $\{\pi_S^{i,j}\}_{S \in \mathbb{S}_I}$ that output* accept *and are partnered with $\pi_T^{q,l}$;*
6. *all oracles $\pi_S^{i,j}$ in the set $\{\pi_S^{i,j}\}$ are partnered with the same authenticator oracle $\pi_T^{q,l}$;*
7. *π_V^h is uniquely partnered with $\hat{\pi}_{C'}^w$;*
8. *the adversary cannot win the indistinguishability game;*
9. *for π_V^h, $a \neq \epsilon$, where ϵ denotes the empty string;*
10. *π_V^h has output accept for identity uid and is partnered with a unique set of at least $|Q|_{min}$ identity server oracles $\{\pi_S^{i,j}\}$ that have output* accept *on the same identity uid;*
11. *$\pi_T^{q,0}$ was created as a consequence of π_T^m accepting command $\langle M_r, t_r \rangle$ under gesture G_r;*
12. *$\pi_T^{q,l}$ was created as a consequence of π_T^p accepting command $\langle M_a, t_a \rangle$ under gesture G_a;*
13. *π_T^m and π_T^p are unique PACA partners of π_C^y and $\pi_{C'}^z$ respectively;*
14. *challenges $\langle rc \rangle$ were produced by $\{\pi_S^{i,0}\}$, received by $\hat{\pi}_C^x$ and used by π_C^y as input to generate $\langle M_r, t_r \rangle$ at a time when π_T^m was valid;*
15. *challenges $\langle ac \rangle$ were produced by $\{\pi_S^{i,j}\}$, received by $\hat{\pi}_{C'}^w$ and used by $\pi_{C'}^z$ as input to generate $\langle M_a, t_a \rangle$ at a time when π_T^p was valid;*
16. *id_S was the server-side input to $\pi_S^{i,0}$ for all $S \in Q$, and id_I such that $\mathsf{id}_S \in \mathsf{id}_I$ was the client-side input to π_C^y and $\pi_{C'}^z$;*
17. *the registration contexts of $\{\pi_S^{i,0}\}$ and $\pi_T^{q,0}$ encode id_I.*

4.11.5 SPOC + CTAP2 security proof

Theorem 3 (SPOC + CTAP2 security)**.** *If* $|Q|_{min} = 2k+1$, $n \in [2k+1, 3k+1]$ *then, for any efficient adversary* $\mathcal{A}$ *against the SPOC+CTAP2 protocol that makes at most* α *queries to* Init*, we can construct adversaries* $\mathcal{E}$ *and* $\mathcal{B}$ *such that:*

$$\mathsf{Adv}^{\mathsf{sps+paca}}_{SPOC+CTAP2}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{uf-t}}_{\mathrm{CTAP2}}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{spc}}_{\mathrm{SWA}}(\mathcal{E}) + \alpha^2 \cdot 2^{-\lambda}$$

The full composition proof builds upon SPOC security proof (Section 4.10) and the composition proof of [16], as Experiment 5 shares similar winning conditions with both.

Proof sketch of Theorem 3. Let's consider the winning conditions of Experiment 5.

(1-3) can be proven as in Section 4.10 proof sketch.

(4, 5) we can prove that 4 and 5 hold with a reduction to the SPC game: we can construct an adversary $\mathcal{E}$ that simulates the SPS+PACA experiment. It uses PACA queries Send, Authorize and Validate along with SPC queries Start, Challenge and Complete to simulate the Start, Challenge and Complete queries of the SPS+PACA experiment. Given that there is no restriction on adversarial queries in the SPC experiment, if 4 or 5 are violated in the SPS+PACA experiment, then they are also violated in the SPC experiment;

(6-10) can be proven as in Section 4.10 proof sketch.

(11-13) given points 1 through 10, it follows that registration and authentication commands $\langle M_r, t_r \rangle$ and $\langle M_a, t_a \rangle$ must have been accepted by authenticator oracles π_T^m and π_T^p, otherwise $\pi_T^{q,0}$ and $\pi_T^{q,l}$ would not have generated any response to terminate registration and authentication session with the corresponding partners $\pi_S^{i,j}$. Furthermore, if 13 does not hold, i.e. π_T^m and π_T^p are not uniquely partnered with π_C^x and $\pi_{C'}^y$ or gestures G_r and G_a do not exist, we can construct an adversary $\mathcal{B}$ against PACA security.

(14, 15) if 14 or 15 do not hold, it means that SPS client oracles $\hat{\pi}_C^x$ and $\hat{\pi}_{C'}^w$, or PACA client oracles π_C^y and $\pi_{C'}^z$ have received different challenges than $\langle rc \rangle$ and $\langle ac \rangle$ from registration and authentication server oracles $\{\pi_S^{i,0}\}$ and $\{\pi_S^{i,j}\}$. However, this would contradict point 4 and thus violate SPC security. This can be handled by $\mathcal{E}$ as a special case.

(16, 17) given the points proved above it follows that the commands accepted by authenticator oracles π_T^m and π_T^p partnered with client oracles π_C^x and $\pi_{C'}^y$ are correct, and thus 16 holds. Moreover, due to SPC partnership proved above, identity server oracles $\{\pi_S^{i,0}\}$ partnered with authenticator oracle $\pi_T^{q,0}$ would not have output accept if the authenticator response was different from the server input. Therefore 17 holds.

□

4.12 Final remarks

This work proposes SPOC, the first survivable passwordless single sign-on protocol. SPOC is practical because it is compatible with standard FIDO2 authenticators available on the market and achieves acceptable performance for modern user experience requirements. We formally demonstrate that this protocol is secure against the same types of adversaries considered by FIDO2 and OIDC. Moreover, we analyze novel attack vectors and introduce new formalizations of security properties that can be of independent interest. This work is open to different research directions, such as investigating which security trade-offs can be obtained by dropping compatibility with standard FIDO2 authenticators, whether it is possible to guarantee a provably secure survivable passwordless SSO which guarantees flexibility in the sense of the protocol described in Chapter 3, or whether it is possible to preserve compatibility with FIDO2 authenticators in a scenario of decentralized governance where identity servers belong to distinct identity providers.

Chapter 5

Scalable, Confidential and Survivable Software Updates

5.1 Introduction

The ability of deploying efficient and secure software updates is one of the most critical aspects of any modern information system. Although update systems have always existed, they do not ensure high security against advanced attacks [69, 92, 97]. Numerous efforts have identified that essential security properties of software updates are *authenticity*, *freshness* and *transparency* [21, 38, 108, 96, 6], and that software update systems must guarantee *availability* and provide *fast*, *resilient* and *scalable* dissemination of software updates to ensure prompt application of security patches [79, 68, 8].

We focus on proprietary software update systems, that impose additional design constraints that do not characterize open source software update systems. In particular, proprietary software update systems must guarantee also *access control* to prevent unauthorized clients from installing unauthorized updates, and *confidentiality* of software updates to protect from reverse engineering. Moreover, we aim at a highly secure system that provides two essential properties, that are *recoverability*, that allows administrators to rapidly recover the system to a safe state after a security incident, and *survivability*, that guarantees security even if the software update system is partially compromised [108, 112, 75, 74, 96]. *Survivability* implies avoiding the presence of single points of failure or vulnerability within the system, making it more difficult for an attacker to compromise software updates in any part of the software development or distribution process.

Previous proposals only satisfied subsets of the mentioned security requirements without presenting a unified solution for the distribution of proprietary software updates [96, 6, 8]. Some proposals focus on open source software, thereby not considering access control and confidentiality requirements [96, 6]. Other work focuses on confidentiality and access control but does not consider survivability and recoverability requirements [8].

To bridge this gap in the literature, we propose a novel comprehensive framework that is able to provide the security guarantees that modern software update systems for proprietary software should have. In particular, we design the first

survivable framework for the secure distribution of confidential updates for proprietary software that satisfies availability, scalability, resiliency, survivability and recoverability requirements, while guaranteeing authenticity, confidentiality, freshness, timeliness and transparency of software updates to clients. The framework enforces fine-grained access control policies over untrusted distribution infrastructures, to comply with distinguished business driven practices. To this aim, our proposal includes two novel contributions of independent interest. First, we extend existing Multi-Authority Attribute-Based Encryption schemes through a novel technique that allows survivable generation of decryption keys so that compromising a threshold of key-generating actors does not allow attackers to violate update confidentiality. Second, we design a novel protocol that allows distributed authentication and encryption of software updates without single points failure. We demonstrate the practicality of the proposed framework through a performance evaluation of our novel Multi-Authority Attribute-Based Encryption extension and distributed authentication and encryption protocol.

The remainder of the chapter is organized as follows. Section 5.2 discusses related work. Section 5.3 describes the system and threat model. Section 5.4 outlines the overall design. Section 5.5 describes the details of each operation. Section 5.6 discusses the security of the proposed system. Section 5.7 evaluates performance and costs. Section 5.8 reports conclusions and future work.

5.2 Related work

This work proposes the first survivable software update framework that integrates all five attributes that should characterize software updates (authenticity, availability, freshness, transparency and confidentiality) and ensures all five guarantees of a software update framework (to be fast, scalable, resilient, survivable and recoverable) that does not use a trusted third party for software distribution. In the following we highlight our original contributions over previous proposals, which involve the attributes of *survivability*, *confidentiality* and *authenticity*.

Confidentiality of software update binaries at rest and in motion is important to protect software updates from automatic exploit generation [35]. To guarantee confidentiality of software updates on untrusted distribution infrastructures, related works adopt different types of encryption schemes. The proposal of [68] makes black-box use of symmetric encryption to encrypt updates with a single symmetric key to allow scalability in the number of clients. The symmetric key is then broadcast to clients to allow decryption. The proposal does not protect the confidentiality of the key during broadcast and therefore is not suitable for proprietary software. The authors in [8] adopt the Ciphertext-Policy Attribute-based Encryption (CP-ABE) scheme proposed in [25] to protect the symmetric key by producing a single ciphertext for all clients. However, in both proposals the key generation procedure of the adopted encryption schemes is not designed to be distributed. This choice represents a single point of failure for the security of the system which, if compromised, would allow the attacker to issue new keys and violate the confidentiality of past and possibly future updates. For this reason, the approaches proposed in [68] and [8] do not guarantee

full survivability. We enhance the survivability of the framework by decentralizing the decryption key generation process by extending the Multi-Authority Ciphertext-Policy Attribute-Based Encryption (MA-CP-ABE) scheme of [105]. In particular, with an appropriate choice of encryption policies we can tolerate the compromise of a threshold of key-generating actors while guaranteeing the confidentiality of previous and future updates. As a result, our proposal is the first survivable software update system which can guarantee confidentiality of software updates. Furthermore, our original extension is fully *recoverable* as, once a compromise is detected, administrators can easily restore the system to a safe state, thereby ensuring service continuity.

Software updates frameworks should allow distribution through untrusted infrastructures, hence it is mandatory to guarantee end-to-end software *authenticity* [21]. The authors of [96] guarantee end-to-end authenticity of software updates through a public and permissioned blockchain that stores authenticated update metadata. However, their proposal cannot guarantee authenticity of confidential software updates because it is designed for open-source software. To guarantee authenticity of confidential software updates, we improve over [96] in multiple ways. First, we design a novel distributed protocol that authenticates encrypted software updates without single points of failure and that allows clients to verify that any update has been approved by a number of authorized actors by means of multi-signatures. Second, we extend the architecture of their proposed blockchain to account for additional roles required to authenticate confidential updates. Moreover, our proposal extends their proposed blockchain to include the due authenticated, survivable and non-equivocable mechanisms and procedures that indicate the software update location to clients. These mechanisms and procedures offer to system administrators the flexibility of choosing and changing the update location and the distribution infrastructure operator as needed. These possibilities are not provided by the authors in [96] that implicitly assume a way of authenticating the update location and do not provide mechanisms to authenticate a location change.

Finally, we integrate our original contributions by extending the architecture and ideas of [96], which allow our proposal to inherit the attributes of *availability*, *freshness*, *timeliness* and *transparency*. Moreover, our proposal guarantees *fast*, *scalabile* and *resilient* dissemination of software updates by adapting the strategy introduced in [8] of producing a single ABE ciphertext per update, to our protocols based on Multy-Authority ABE.

5.3 System and threat model

5.3.1 System model

The typical scenario for proprietary software update systems involves three entities: *software house*, *(software) distribution infrastructures* and *clients*.

The *software house* includes a set of roles that share the same interests. Within the software house, we denote as *developers* a set of employees that can access source code, compile it, produce software binaries, and are responsible for approving new software versions that are identified by increasing alphanumerical

strings. Depending on the characteristics of the software house, this role can be accomplished by actual developers or by other specialized personnel, such as that dedicated to DevOps practices. The software house approves updates that must be delivered to clients. Each update consists of binaries and related source code.

The software house relies on *distribution infrastructures*, which could be managed by third parties such as Content Delivery Networks (CDN) or community-managed mirror servers. The software house also defines access control policies over software updates which are enforced by a trusted distribution infrastructure.

Finally, clients represent the devices that store and execute the version of the installed software binaries produced and maintained by the software house. Clients periodically query the distribution infrastructure to check whether a new update is available and, if so, they can download it from the distribution infrastructure.

In the following we describe our proposal's system model. We extend the reference software house by adding three roles: *admins*, *authentication server* and *Attribute-based Encryption* (ABE) servers. We consider a software house with n_d developers, n_a admins, n_r ABE servers and one authentication sever. Each developer $d \in [n_d]$ has a signing key pair $dk_d = \langle sk_d, pk_d \rangle$ and has access to the software source code (src). Admins are responsible for managing security-critical cryptographic material for authenticating roles and enforcing access control policies. Each admin $a \in [n_a]$ has a signing key pair $ak_a = \langle sk_a, pk_a \rangle$ and has access to the access control policies ($\mathbb{P}$) that must be used for encrypting software update binaries through MA-CP-ABE. Authentication and ABE servers are responsible for managing cryptographic material. ABE servers are authorities responsible for issuing ABE keys to authenticated clients for decrypting software updates. Each ABE server $r \in [n_r]$ has an ABE authority key pair $rk_r = \langle sk_r, pk_r \rangle$ and the set $\mathcal{A}_{\zeta(r)}$ of attributes to compute the decryption keys for clients. The authentication server maintains the database of registered clients and the corresponding authentication information, and assigns attributes to clients after a successful authentication.

The distribution infrastructure maintains encrypted software updates (encrypted binaries) associated with location information that is used by other parties to retrieve updates. Our proposal relies on untrusted distribution infrastructures by making use of Multi-Authority Ciphertext-Policy Attribute-Based Encryption (MA-CP-ABE) [105], which guarantees confidentiality and enforces policy-based access control over software updates even on untrusted distribution infrastructures. In this scheme an authority issues to clients one or more private keys each encoding an attribute. The encryption algorithm accepts a message and a policy expressed as a monotonic boolean formula over attributes, and produces a ciphertext. A client is able to decrypt the ciphertext if the attributes of his private keys satisfy the boolean formula associated to the ciphertext. To this aim, each registered client has a set SK_{cid} of ABE keys for decrypting software updates.

Finally, our system requires two additional roles that are inherited from

the scenario in [96]: *validators* and *witnesses*. Validators audit and validate new software updates through reproducible builds. Witnesses are nodes of a *multi-layer skipchain* which is an authenticated append-only data structure introduced by [96], that stores public cryptographic material and software update metadata. Witnesses share the same version of the multi-layer skipchain by using a Byzantine-fault-tolerant state-machine-replication consensus algorithm.

The proposed system considers n_v validators and n_w witnesses. Each validator $v \in [n_v]$ has a signing key pair $vk_v = \langle sk_v, pk_v \rangle$ and a signed copy of the source code for update validation. Each witness $w \in [n_w]$ has a signing key pair $wk_w = \langle sk_w, pk_w \rangle$ and maintains a copy of the *multi-layer skipchain*.

5.3.2 Threat model

We consider an attacker that may be interested in violating confidentiality, authenticity, availability or integrity of software updates. Violating confidentiality means that the attacker can reverse engineer the update and look for vulnerabilities in the previous or in the update version. Compromising the authenticity and integrity of updates may induce clients to download and install backdoored software versions. Denying an update forces a client to keep an outdated software version which may contain vulnerabilities that an attacker can exploit.

Our proposal protects software updates binaries and clients against the mentioned attacks by using cryptographic protocols, and assuming a computationally bound attacker which is unable to break the security of the adopted protocols or the security of their underlying cryptographic primitives.

We inherit the following threshold assumptions from [96]. We assume that all actors communicate over authenticated channels that can be eavesdropped by the adversary. Survivability is guaranteed through threshold variants of cryptographic schemes.

We assume that no more than a threshold of k_d out of n_d developers is malicious. A developer is malicious if he colludes with the attacker, if his signing key has been compromised or if the attacker has compromised other parts of the developers' systems, such as by covertly installing a compromised compiler.

We assume that no more than a threshold k_a out of n_a *admins* is malicious and that no more than k_r out of n_r *ABE servers* is malicious. For simplicity we assume that the *authentication server* is honest and that the authentication mechanism adopted to authenticate clients is secure. We could relax these simplifying assumptions by adopting survivable authentication mechanisms such as the ones described in Chapters 3 and 4 to remove single points of failure in client authentication phases.

We assume that no more than k_v *validators* and no more than $k_w = \lfloor n_w/3 \rfloor$ *witnesses* are malicious. These thresholds protect the correctness of data inserted by admins in the multi-layer skipchain and the security of its consensus mechanism, that is executed by witnesses. Moreover, we assume that validators do not leak source code. Indeed, since validators must receive the project source code in plaintext form to validate it, developers must trust all validators not to collude with the adversary. The validator role represents a trade-off between source code confidentiality and transparency of software updates. To the best of

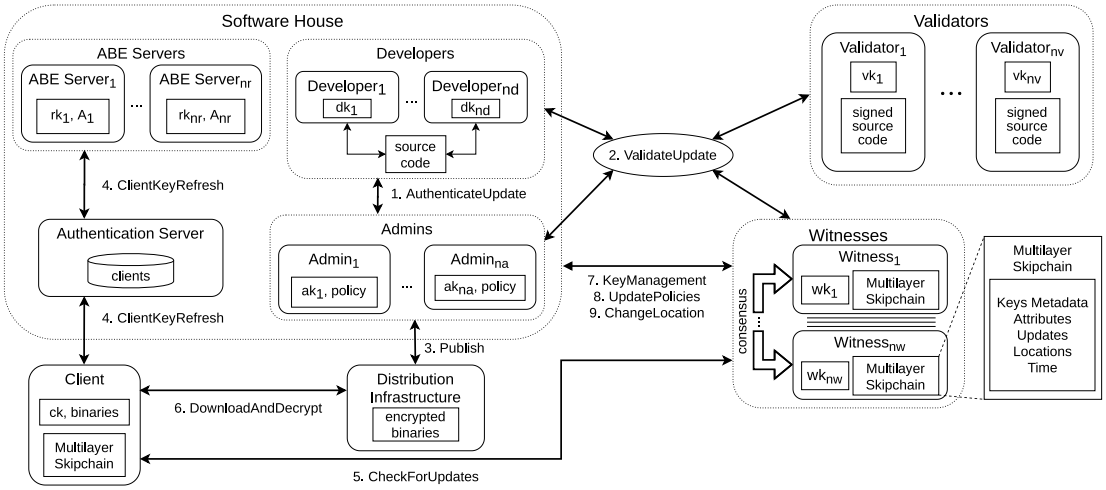

Figure 5.1: Architecture of the framework for secure software updates

our knowledge, enabling software transparency without relying on third parties is still an open problem.

The distribution infrastructure may be untrusted that is, attackers can replace, eavesdrop and modify software updates during its distribution, but they cannot impede availability for an unbounded amount of time.

A client with proper attributes can decrypt the update, and an attacker that compromises or disguises as a client can obtain access to the decrypted update and reverse engineer it. To protect against malicious clients, it is possible to adopt orthogonal solutions, such as patch obfuscation, to thwart reverse engineering of the released binary. However, protections against similar threats are out the scope of this proposal. We assume that the client is honest and that an attacker cannot break the update confidentiality by compromising or impersonating a client.

5.4 Framework design

5.4.1 Framework components and operations

We describe the proposed framework for secure software updates by referring to Figure 5.1. This figure represents the components of the system and its main operation flows. The secure software distribution framework includes nine operations: (1) update authentication, (2) update validation, and (3) publish are used by the software house members to make a new update available for clients. The operations (5) check for update and (6) download and decrypt update are used by clients to detect and obtain new released updates, and possibly detect attacks. The other operations are used for additional security-related tasks: (4) client key refresh allows clients to obtain new decryption keys; (7) key management, (8) update policies and (9) change location allow admins

to rotate public keys, ABE attributes and update the location of the encrypted binaries, respectively.

These operations are executed as follows. When a new update is ready, admins and developers authenticate it (1) and send the authenticated update to validators that check the source-to-binary correspondence and return an authenticated validity attestation (2). Admins append to the multi-layer skipchain the authenticated metadata and update attestations, and publish the authenticated binaries (3) to the distribution infrastructure. A client, who has already obtained a valid set of ABE keys (4), checks whether new updates are available (5) by querying the multi-layer skipchain. If available, the client downloads them from the distribution infrastructure and decrypts them with his ABE keys (6). In case the new update requires a policy change, admins execute the update policies procedure (8) before authenticating the update. If keys of any role need to be rotated because of a security incident or because of key expiration, admins execute the key management procedure (7). If admins need to change the location of the latest update, for example due to change in distribution infrastructure provider, admins execute the change location procedure (9).

In the following two subsections, we outline the operations of the multi-layer skipchain and of the information flow, respectively. Details of each operation are described in Section 5.5.

5.4.2 Multi-layer skipchain

The witnesses maintain a multi-layer skipchain that guarantees freshness and non-equivocation of software update metadata to clients and that allows admins survivable and authenticated modification of the corresponding cryptographic material. Our original design extends that proposed in [96] and consists of eight layers shown in Figure 5.2. Layers are stacked in the following order and are identified by labels: admins ($\mathsf{L_{ad}}$), witnesses ($\mathsf{L_w}$), validators ($\mathsf{L_v}$), ABE servers ($\mathsf{L_r}$), attributes ($\mathsf{L_{at}}$), update ($\mathsf{L_u}$), location ($\mathsf{L_l}$), time ($\mathsf{L_t}$). Each layer has specific constraints that newly appended data must satisfy.

The first four layers (admins, witnesses, verifiers, ABE servers) store public keys of admins, witnesses, verifiers, and ABE servers, respectively. They allow admins to rotate the public keys of these actors. We collectively refer to these layers as *keys metadata* layers. A new set of public keys can be added to any of the keys metadata layers only if it is authenticated by at least a threshold of admins.

The attributes layer maintains the set of ABE attributes that are adopted in the access control policy used to encrypt the latest update with ABE. A new set of attributes can be added to this layer only if the set is authenticated by at least a threshold of admins. This layer allows admins to change the set of valid attributes when needed, and allows clients to receive the list of valid ABE attributes to determine if they need to refresh their ABE keys (for details about ABE key refresh, see Section 5.5.6). We note that admins cannot change the set of ABE attributes of already released encrypted binaries because the skipchain is append-only. Any change would have no effect on existing ciphertexts already

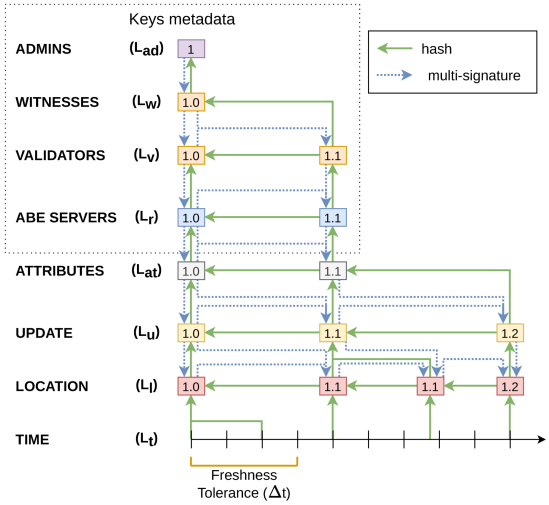

Figure 5.2: The multi-layer skipchain for secure software updates

published in the distribution infrastructure. A policy change, and consequently an attribute change, has effect only on future updates.

The update layer maintains metadata about encrypted binaries. New update metadata can be added to the update layer only if they have been authenticated by a threshold of admins, developers and validators. In particular, validators are in charge of verifying the correspondence between source code, encrypted binaries and metadata (see Section 5.5.5).

The location layer maintains authenticated location information, such as a URL, about how to retrieve encrypted update binaries from the distribution infrastructure. New location information can be added to the location layer only if it is authenticated by at least a minimum threshold of admins. This layer allows admins to change the update location when necessary.

The time layer maintains multi-signed, timestamped hash pointers to the location layer, that are periodically computed by witnesses. A new timestamp can be added to the time layer only if it is authenticated by at least a minimum threshold of witnesses and only if the timestamp is greater than the previous timestamp, and not too far into the future. Assuming that a client has a trusted and reliable time source, this layer allows clients to detect freeze attacks by comparing the most recent timestamp with the client's current timestamp, and verifying that the difference between the two timestamps is within an upper bound (*freshness tolerance*) that is determined by the admins in the update metadata. We assume that a client can safely bootstrap his copy of the multi-layer skipchain by securely obtaining the admins public keys included in the genesis skipblock of the admins layer.

5.4.3 Information flow

We describe the main operations supported by the system. Details of each operation are reported in Section 5.5 and 5.6.

Setup. The goal of the setup operation is to authenticate the public keys of all actors in the system. This operation is orchestrated by admins who act as trust anchor. Each admin, witness, validator, ABE server and developer generates a key pair and maintains the secret key confidential. Then, admins act as certification authorities and gather the public keys of witnesses, validators and ABE servers. They collectively authenticate them through multi-signatures and by assigning validity periods that produce key metadata. Finally, admins store key metadata in the appropriate skipchain layers.

Update policies. When the software house defines a new access control policy or modifies an existing policy or when admins rotate the ABE servers keys, admins compute a new set of attributes and assign each attribute to an ABE server. Admins communicate the new set of attributes to the authentication server and append the new set to the attributes skipchain layer.

Update authentication. When a new software update is ready to be released, the source code must be approved by validators. To this aim, developers and admins must issue to them the source code together with cryptographic material that assesses the compilation and encryption procedures used to produce the encrypted binaries distributed to clients. The overall procedure includes five phases. A threshold of admins authenticates the access control policy ($\mathbb{A}$), the freshness tolerance value that must be enforced on the released software, and the set of developers that are authorized to sign the update source code. Then, the same admins communicate the authenticated access control policy, the freshness tolerance and the developers public keys to each authenticated developer. After these operations, a threshold of developers compiles the source code through reproducible builds procedures, and all of them obtain the same binary data. To obtain the same encrypted binaries, the same developers collaboratively generate a shared secret cryptographic key and encrypt the resulting software binaries through a deterministic symmetric encryption procedure. The developers also compute a digest of the encrypted binaries and ABE-encrypted key. Finally, the involved developers use a multi-signature scheme to authenticate the source code and the generated public cryptographic material.

Update validation. Admins send the authenticated source code, the encryption key and authentication material to validators for validation. Validators compile the source code through reproducible build procedures and encrypt it through deterministic encryption using the received encryption key. Then, they verify the correctness of the resulting encrypted binaries against the received authentication material. If verification is valid, the validators apply a multi-signature to the received authentication material by interacting with admins according to the required validation phase of the consensus protocol used by witnesses. Finally, admins send the location information, the authentication material and the validators multi-signature to witnesses to update the multi-layer skipchain.

Publish. Any admin can publish the encrypted software binaries to the distribution infrastructure at the location inserted in the multi-layer skipchain. We observe that the encrypted binaries could also be published before the completion of the validate update algorithm to ensure the availability of the update to clients as soon as the multi-layer skipchain is updated.

Client key refresh. To decrypt the encrypted binaries, a client must obtain valid ABE keys from ABE servers of the software house. We describe the procedure by assuming, without loss of generality, that clients are already registered in the authentication server's clients database. Each client authenticates himself at the authentication server. This server determines the ABE attributes that qualify it and sends them to a threshold of ABE servers. ABE servers respond with ABE keys and the authentication server forwards them to the client. We highlight that the keys obtained by a client are related to the access control policy used to encrypt binaries, and can be reused for unlimited software releases as long as admins do not operate key rotations that invalidate the client decryption keys, or use new attributes for encrypting new software releases that are mandatory to decrypt binaries. Only in these cases, and only if the client still complies to the software house policies, a client must obtain new ABE keys by re-executing the client key refresh procedure.

Check for updates. Clients periodically check for updates by requesting to witnesses the last skipchain updates. Clients must always be able to obtain a response from witnesses, and the response must always include updated timestamps of the time layer. A missing response or a response with stale or old time information are considered as violations of the system availability, possibly due to ongoing attacks. Clients verify the authenticity and integrity of the received skipblocks by using admins and witnesses public keys, and verify the authenticity of skipblocks payloads by using admins, validators and developers public keys. A new software release is available if and only if there is a new authentic skipblock in the update layer. In this case, clients extract the most-updated location information and can proceed to obtain encrypted binaries.

Download and decrypt. Clients use the obtained location information to query the distribution infrastructure for the encrypted binaries. We assume that the distribution infrastructure is always available and responds to client queries. Clients download the encrypted binaries, verify their authenticity and decrypt them by using ABE decryption keys to obtain plaintext software binaries. At this point, the clients can install the software update binaries.

Key management. Occasionally, admins may have to rotate, revoke or issue new keys to substitute compromised keys, replace or remove misbehaving actors or add new actors to the system. Admins rotate, revoke and issue keys by collectively authenticating a new set of public keys and sending it to witnesses that update the appropriate keys metadata layer.

Change location. Finally, admins may need to change the location of an update. Admins can notify clients about the location change by appending new location information in the corresponding layer of the skipchain. Thanks to the design of the skipchain layers, this operation does not require any modification to the update metadata and can be operated without the intervention of the

validators.

5.5 Framework Details

We describe the details of the proposed framework. For simplicity, we assume that the parameters of the adopted schemes are already defined, including elliptic curves, symmetric ciphers and different types of hash functions (standard hash functions and those required to produce elliptic curve points). Moreover, we omit technicalities such as translations between monotonic boolean formulas, used in this work, and monotone span programs, on which ABE schemes are based (see [18] for details).

5.5.1 Adopted schemes and notations

We describe the operations frameworks of the adopted Multi-Authority Ciphertext-Policy Attribute-Based Encryption scheme (MA-CP-ABE) [105], multi-signatures scheme [30] and multi-layer skipchain (see Section 5.4.2).

MA-CP-ABE includes the following algorithms:

$\langle sk_r, pk_r \rangle \leftarrow \mathsf{AuthSetup}(r)$: is a randomized algorithm executed by each authority $r \in \mathcal{R}$ where $\mathcal{R}$ is the set of authorities identifiers. The algorithm returns a key pair $\langle sk_r, pk_r \rangle$.

$sk \leftarrow \mathsf{KeyGen}(cid,\ sk_r,\ e_r)$: is a randomized algorithm that takes a client global unique identifier cid, authority's r secret key sk_r and an attribute e_r. The algorithm is executed by the authority r responsible for attribute e_r and returns a secret key sk for attribute e_r to the client identified by the global unique identifier cid.

$c \leftarrow \mathsf{Encrypt}_{\mathsf{ABE}}(m, \mathbb{A}, \{pk_r\})$: is a randomized algorithm that takes a message m, an access-control policy $\mathbb{A}$ expressed as a monotonic boolean formula over attributes, and the set of public keys $\{pk_r\}$ of the authorities that control at least an attribute in the policy. The algorithm, which can be publicly executed, returns the ciphertext c.

$m \leftarrow \mathsf{Decrypt}_{\mathsf{ABE}}(\{sk\},\ c)$: is a deterministic algorithm run by a client that takes the set of secret keys $\{sk\}$ and a ciphertext c. The algorithm returns plaintext m only if the attributes associated with the client's secret keys satisfy the ciphertext policy.

Multi-signatures allow any subgroup L of a group of n players $\{V_1, \ldots, V_n\}$ to collectively sign a message m and prove to a verifier that all members of L participated in producing the message signature. We consider using the scheme proposed in [30] based on BLS signatures [31], that includes the following three algorithms.

$\langle sk_i, pk_i \rangle \leftarrow \mathsf{KeyGen}(1^\lambda)$: is a randomized key generation algorithm run by each player V_i that returns key pair $\langle sk_i, pk_i \rangle$.

$\langle L, \sigma, m \rangle \leftarrow \mathsf{MultiSign}(\{sk_i\}_{i \in L},\ m)$: is a possibly randomized two-step interactive algorithm run by any subset of players $L \subseteq \{V_1, \ldots, V_n\}$.

$b \leftarrow \mathsf{Verify}(\{pk_i\}_{i \in L}, \sigma, m)$: is a deterministic algorithm run by the verifier and outputs $b = 1$ if and only if signature σ has been generated with $\mathsf{MultiSign}(\{sk_i\}_{i \in L},\ m)$, 0 otherwise.

Multi-layer skipchain includes the following algorithms:

$\sigma \leftarrow \mathsf{Append}(l, d)$: is an interactive algorithm executed between a client and the nodes maintaining the multi-layer skipchain. The algorithm, started by the client, takes a layer identifier l and the data d the client wants to append to layer l. If data d satisfies the constraints of layer l, then d is added as payload of layer's l new skipblock and nodes return to the client a multi-signature σ of the pair $\langle l, d \rangle$ to confirm that d has been appended to layer l.

$(\pi, S) \leftarrow \mathsf{GetLatestSkipblocks}(t)$: is an interactive algorithm executed between the client and the nodes maintaining the multi-layer skipchain. The algorithm, started by the client, takes the latest timestamp t indicating the last time the client updated the multi-layer skipchain, and returns the set of skipblocks S the client lacks, and a proof π that the set S is valid and fresh. In particular, the proof π contains the latest timestamp t' of the *time* layer, and the minimum set of forward pointers and *witnesses* layer skipblocks required to validate the set S.

$0, 1 \leftarrow \mathsf{Validate}(\mathsf{now}, \langle \pi, S \rangle)$: is an algorithm executed by the client that takes the client's current time, denoted as now, and the pair $\langle \pi, S \rangle$ obtained by the client with $\mathsf{GetLatestSkipblocks}(\cdot)$. The algorithm returns 1 if the proof π for skipblocks S is valid and if the difference between now and the proof timestamp t' is within a certain threshold, 0 otherwise. We note that the threshold value may be an application-defined value included in skipblocks payload.

5.5.2 Setup

We model the setup operation as:

$$\mathsf{Setup}(1^\lambda) \tag{5.1}$$

where 1^λ is the security parameter.

Key metadata attestations are attached to multi-signatures to demonstrate the chain of trust between signers and *admins*, and include metadata established by *admins* to define authenticity threshold requirements. We define the *Key Metadata Attestation* (KMA) as follows:

$$\mathsf{KMA} = \langle \langle \{pk\}, s, v_b, v_e \rangle, k_a, \sigma_{\mathsf{KMA}} \rangle \tag{5.2}$$

$$\begin{aligned} \sigma_{\mathsf{KMA}} \leftarrow \mathsf{MultiSign}(SK,\ & \langle \langle \{pk\}, s, v_b, v_e \rangle, k_a \rangle), \\ & SK \subseteq \{sk_a\} : |SK| > k_a \end{aligned} \tag{5.3}$$

where s is the threshold on the minimum number of public keys in $\{pk\}$ used to verify multi-signatures, v_b and v_e denote the begin and end of the attestation validity period and k_a is the threshold of tolerable malicious *admins*. KMA is valid only if σ_{KMA} is a valid multi-signature computed by at least $k_a + 1$ *admins*, and $v_b < v_e$.

Each *admin* generates his multi-signature key pair:

$$\langle sk_a, pk_a \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{5.4}$$

Admins compute the *admins* KMA (AKMA), which includes the *admins* public keys, as following:

$$\mathsf{AKMA} = \left\langle \left\langle \{pk_a\}_{a \in [n_a]}, k_a, v_b, v_e \right\rangle, k_a, \sigma_{\mathsf{AKMA}} \right\rangle \tag{5.5}$$

We note that, in this particular case, AKMA has the further constraint of requiring verification by using a subset of the public keys included in the attestation itself, similarly to self-signed root certificates in PKI systems. We highlight that the signing operation denotes a distributed protocol for computing the multi-signature among mutually untrusted parties, in this case *admins*.

Witnesses generate their multi-signature signing keys:

$$\langle sk_w, pk_w \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{5.6}$$

and send their public keys $\{pk_w\}$ to *admins*. *Admins* compute the *witnesses* KMA (WKMA), which includes the *witnesses* public keys, as following:

$$\mathsf{WKMA} = \left\langle \left\langle \{pk_w\}_{w \in [n_w]}, k_w, v_b, v_e \right\rangle, k_a, \sigma_{\mathsf{WKMA}} \right\rangle \tag{5.7}$$

where $k_w = \lfloor n_w/3 \rfloor$.

Validators generate their multi-signature signing keys:

$$\langle sk_v, pk_v \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{5.8}$$

and send their public keys $\{pk_v\}$ to *admins*. *Admins* compute the *validators* KMA (VKMA), which includes the *validators* public keys, as following:

$$\mathsf{VKMA} = \left\langle \left\langle \{pk_v\}_{v \in [n_v]}, k_v, v_b, v_e \right\rangle, k_a, \sigma_{\mathsf{VKMA}} \right\rangle \tag{5.9}$$

In the following we denote as $r \in \mathcal{R}$ the identifier of an *ABE server*, where $\mathcal{R}$ is the set of identifiers of all *ABE servers*. Each *ABE server* generates his key pair $\langle sk_r, pk_r \rangle$:

$$\langle sk_r, pk_r \rangle \leftarrow \mathsf{AuthSetup}(r) \ \forall r \in \mathcal{R} \tag{5.10}$$

All *ABE servers* send their public keys $\{pk_r\}$ to *admins*. *Admins* compute the *ABE servers* KMA (RKMA), which includes the *ABE servers* public keys, as following:

$$\mathsf{RKMA} = \left\langle \left\langle \{pk_r\}_{r \in \mathcal{R}}, k_r, v_b, v_e \right\rangle, k_a, \sigma_{\mathsf{RKMA}} \right\rangle \tag{5.11}$$

Any *admin* initializes the appropriate skipchain layer by executing $\mathsf{Append}(\cdot, \cdot)$, as described in Section 5.5.1, using all metadata obtained so far (AKMA, WKMA, VKMA, RKMA). The *admin* then verifies that the returned multi-signatures are valid and that the set of signers is a subset of the *witnesses* specified in WKMA.

Developers generate their multi-signature signing keys:

$$\langle sk_d, pk_d \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{5.12}$$

and send their public keys $\{pk_d\}$ to *admins*. We note that *developers* public keys are authenticated during the *update authentication* procedure (see Section 5.5.4).

We highlight that *admins*, *developers*, *witnesses* and *verifiers* also generate the due cryptographic keys to establish authenticated and confidential point-to-point communication channels, and to generate the required cryptographic material in following phases. For ease of exposition we do not specify their generation and usage as we rely on well known cryptographic primitives.

5.5.3 Update policies

We model the update policies procedure as $\mathsf{UpdatePolicy}(\mathbb{P}, v, k_r, \mathcal{R})$, where $\mathbb{P}$ denotes the access control policy, v denotes the minimum version number to which the policy applies, k_r denotes the security threshold for *ABE servers*, and $\mathcal{R}$ is the set of *ABE servers*.

The procedure must be used before releasing the first software update, before releasing a new software update that requires a novel policy, or whenever *ABE servers* public keys are rotated. It transforms access control information defined by the *software house* based on a single-authority paradigm to the multi-authority setting of the proposed architecture. It is composed of two phases:

- *attribute derivation* transforms the access control attributes and produces an *authenticated attributes matrix* $\mathcal{A}$ that assigns multi-authority attributes to each ABE server. The matrix $\mathcal{A}$ is also appended to the *attributes* layer of the multi-layer skipchain to be available for *clients* to detect whether a key refresh is needed (Section 5.5.6);
- *policy translation* transforms the single-authority policy $\mathbb{P}$ into the multi-authority policy $\mathbb{A}$, which is used in the *update authentication* procedure (Section 5.5.4).

We observe that *attribute derivation* must be operated only in case of policies modifications that use novel attributes or in case of ABE servers key rotations. Moreover, we note that introducing novel attributes and rotating ABE servers keys does not require re-executing the setup procedure because the ABE scheme adopted in our proposal does not fix the set of ABE servers and attributes during its setup (see Section 5.5.1 for details about the ABE scheme, and Section 5.5.9 for details about ABE servers key rotation). This is a very important property because it enables recoverability as we discuss in Section 5.6. We highlight that, to the best of our knowledge, the proposed approach in *attribute derivation* and *policy translation* procedures is the first practical solution that enables a survivable generation of ABE keys.

Attribute derivation. *Admins* receive the set of the attributes that are used by the *software house* to define the access control policy $\mathbb{P}$. In the following, we denote these attributes as *original attributes* for disambiguation, and we model them as binary strings of potentially variable length. To guarantee survivability, all *ABE attributes* of each *ABE server* must be associated to *original attributes* in a bijective relation. To this aim, *admins* enumerate all of *original attributes* in an ordered set that we denote as P (e.g., by sorting them with lexicographic comparisons). We denote as $p_j \in P$ the j^{th} *original attribute*, where $j \in [|P|]$.

Given the set P of all *original attributes* and the set $\mathcal{R}$ of all *ABE servers*, a selected *admin* computes the set of *ABE attributes* assigned to each *ABE server* as following. To this aim, he generates the *ABE attributes matrix* $\mathcal{A}$ of size $|\mathcal{R}| \times |P|$. Each *original attribute* p_j is mapped to column j. Moreover, we assume that a function $\zeta(\cdot) : \mathcal{R} \to [|\mathcal{R}|]$ exists to map each *ABE server* $r \in \mathcal{R}$ to a row of the matrix. Each element $\alpha^j_{\zeta(r)}$ in the matrix $\mathcal{A}$ is computed as

the concatenation of the public key of the row's *ABE server* with the column's *original attribute*:

$$e_{r,j} := pk_r || p_j, r \in \mathcal{R}, j \in [|P|] \tag{5.13}$$

$$\mathcal{A} := \left(\alpha^j_{\zeta(r)} : \alpha^j_{\zeta(r)} \leftarrow e_{r,j}\right), \forall r \in \mathcal{R}, j \in [|P|] \tag{5.14}$$

Uniqueness of public keys implies uniqueness of *ABE attributes*. All *ABE attributes* that differ only for the public key part are syntactically different and semantically equivalent. The j^{th} column of $\mathcal{A}$, denoted as $\mathcal{A}^j$, contains all semantically equivalent representations of *original attribute* p_j assigned to different *ABE servers*. The $\zeta(r)^{th}$ row of $\mathcal{A}$, denoted as $\mathcal{A}_{\zeta(r)}$, contains all *ABE attributes* of *ABE server* r.

Admins compute the *Authenticated Attributes Map* (AAM) as:

$$\mathsf{AAM} := \langle \langle \mathcal{A}, v \rangle, \sigma_{\mathsf{AAM}} \rangle \tag{5.15}$$

where v is the update version and σ_{AAM} is *admins* multi-signature on tuple $\langle \mathcal{A}, v \rangle$. A designated *admin* writes AAM to the *attributes* layer of the multi-layer skipchain, by executing $\mathsf{Append}(\mathsf{L_{at}}, \mathsf{AAM})$. Witnesses append AAM only if σ_{AAM} is valid.

Finally, a designated *admin* sends the pair $\left\langle \left\langle \mathcal{A}_{\zeta(r)}, v \right\rangle, \sigma_r \right\rangle$ to *ABE server* r, $\forall r \in \mathcal{R}$. Each *ABE server* obtains the latest version of *admins* public keys from the *admins* skipchain layer and accepts the pair $\left\langle \left\langle \mathcal{A}_{\zeta(r)}, v \right\rangle, \sigma_r \right\rangle$ only if σ_r is valid.

Policy translation. In this phase, *admins* translate access control policy $\mathbb{P}$ into a semantically equivalent policy $\mathbb{A}$ expressed over the *ABE attributes* computed in the previous *attribute derivation* phase. Without loss of generality, we describe the translation phase by representing access control policies $\mathbb{P}$ and $\mathbb{A}$ as boolean formulas expressed over *original* and *ABE attributes*, respectively. The boolean formula representing $\mathbb{P}$ must be translated so that satisfying a threshold of $k_r + 1$ semantically equivalent *ABE attributes* implies satisfying the corresponding *original attribute*. We recall that k_r is the maximum amount of malicious *ABE servers*. A designated *admin* translates the original access control policy by substituting each *original attribute* p_j with a boolean expression that returns *true* only if at least a threshold of $k_r + 1$ *ABE attributes* that are semantically equivalent to the *original attribute* are *true*. To this aim, the *admin* computes the set $\mathbb{S}_j$ of all possible subsets S_j of $\mathcal{A}^j$ of cardinality equal to $k_r + 1$, that is:

$$\mathbb{S}_j := \left\{ S_j : S_j \subseteq \mathcal{A}^j, |S_j| = k_r + 1 \right\} \tag{5.16}$$

where n_r is the total number of *ABE servers* and is greater than k_r. The resulting boolean formula $\mathbb{A}$ is computed by substituting each attribute p_j of the original boolean formula $\mathbb{P}$ as following:

$$p_j \leftarrow \bigvee_{S_j \in \mathbb{S}_j} \left(\bigwedge_{e \in S_j} e \right), \forall p_j \in \mathbb{P} \tag{5.17}$$

Example. To clarify the *update policies* procedure we propose an example, where we consider a scenario in which a new software update for "premium" users who have paid a subscription fee is about to be published. The *software house* defines the access control policy $\mathbb{P}$ as the following formula:

$$\text{“}\mathsf{premium}\text{”} \wedge \text{“}\mathsf{paid}\text{”} \tag{Ex. 1}$$

We assume that *admins* have configured three *ABE servers* ($\mathcal{R} = \{r_1, r_2, r_3\}$), and that they want to ensure 1-out-of-3 survivability, that is, tolerating the compromise of one *ABE server*. *Admins* extract and enumerate attributes of Formula (Ex. 1) obtaining "premium" and "paid" ($P = \{\text{“}\mathsf{premium}\text{”}, \text{“}\mathsf{paid}\text{”}\}$). In the following we use the binary operator $\|$ to denote the concatenation of the binary representation of the operands. The attribute matrix $\mathcal{A}$ is:

ABE servers	Attributes	
	premium	paid
r_1	“$pk_{r1}\|\|\mathsf{premium}$”	“$pk_{r1}\|\|\mathsf{paid}$”
r_2	“$pk_{r2}\|\|\mathsf{premium}$”	“$pk_{r2}\|\|\mathsf{paid}$”
r_3	“$pk_{r3}\|\|\mathsf{premium}$”	“$pk_{r3}\|\|\mathsf{paid}$”

Table 5.1: Example attribute matrix $\mathcal{A}$

In Formula Ex. 2 we represent semantically equivalent attributes of Table 5.1 with the original attribute name and with the row index as subscript. *Admins* can finally translate formula Ex. 1 with the following semantically equivalent formula:

$$\begin{aligned}
&(\text{“}pk_{r1}\|\mathsf{premium}\text{”} \wedge \text{“}pk_{r2}\|\mathsf{premium}\text{”}) \vee \\
&(\text{“}pk_{r2}\|\mathsf{premium}\text{”} \wedge \text{“}pk_{r3}\|\mathsf{premium}\text{”}) \vee \\
&(\text{“}pk_{r1}\|\mathsf{premium}\text{”} \wedge \text{“}pk_{r3}\|\mathsf{premium}\text{”}) \\
&\qquad\qquad \wedge \\
&\quad (\text{“}pk_{r1}\|\mathsf{paid}\text{”} \wedge \text{“}pk_{r2}\|\mathsf{paid}\text{”}) \vee \\
&\quad (\text{“}pk_{r2}\|\mathsf{paid}\text{”} \wedge \text{“}pk_{r3}\|\mathsf{paid}\text{”}) \vee \\
&\quad (\text{“}pk_{r1}\|\mathsf{paid}\text{”} \wedge \text{“}pk_{r3}\|\mathsf{paid}\text{”})
\end{aligned} \tag{Ex. 2}$$

5.5.4 Update authentication

We model the update authentication procedure as $\mathsf{AuthenticateUpdate}(\mathsf{src}, \{pk_d\}, v, \mathbb{A}, \Delta t)$, where src is the update source code, $\{pk_d\}$ is the set of *developers* public keys authorized to authenticate src, v is the update version, $\mathbb{A}$ is the multi-authority policy and Δt is the freshness tolerance value. The goal of this phase is to compute two categories of authenticated update metadata: *Authenticated Update Validation Metadata* (AUVM), intended to be used by *validators* in the

update validation phase (Section 5.5.5), and *Authenticated Binaries Metadata* (ABM), intended to be used by *clients* during update retrieval (Section 5.5.7 and Section 5.5.8). This procedure includes two phases operated by *admins* and *developers*, respectively.

Admins bind the update version to the *multi-authority policy* $\mathbb{A}$ and to the *freshness tolerance value* Δt by multi-signing tuples $\langle \mathbb{A}, v \rangle$ and $\langle \Delta t, v \rangle$, producing signatures $\sigma_{\mathbb{A}v}$ and $\sigma_{\Delta tv}$:

$$\sigma_{\mathbb{A}v} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \mathbb{A}, v \rangle) \tag{5.18}$$

$$\sigma_{\Delta tv} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \Delta t, v \rangle) \tag{5.19}$$

We observe that the two bindings are computed separately because they must be verified in different procedures. *Admins* compute the *developers* KMA (DKMA), which includes the *developers* public keys $\{pk_d\}$ authorized to authenticate the update at version v, as following:

$$\mathsf{DKMA} = \left\langle \left\langle \{pk_d\}_{d \in [n_d]}, v, k_d \right\rangle, k_a, \sigma_{\mathsf{DKMA}} \right\rangle \tag{5.20}$$

We note that the DKMA attestation has the update version v in place of the validity period bounds v_b and v_e defined in KMA because the set of keys $\{pk_d\}$ is valid only for version v.

Admins send DKMA and the tuples $\langle \sigma_{\mathbb{A}v}, \mathbb{A}, v \rangle$ and $\langle \sigma_{\Delta tv}, \Delta t, v \rangle$ to each *developer* who verifies the multisignatures $\sigma_{\mathbb{A}v}$ and $\sigma_{\Delta tv}$.

A subset $D \subseteq [n_d]$ such that $|D| > k_d$ of developers authorized in DKMA participates in the following operations. Each *developer* in D builds through reproducible builds procedures the update source code src, obtaining the update binaries bin:

$$\mathsf{bin} \leftarrow \mathsf{DeterministicBuild}(\mathsf{src}) \tag{5.21}$$

Developers agree on a shared deterministic encryption key ψ by using an authenticated group key agreement [34]:

$$\psi \leftarrow \mathsf{KeyAgree}(1^{\mu}, \{pk_d\}_{d \in D}) \tag{5.22}$$

where 1^{μ} is the security parameter. Each participating *developer* encrypts the update binaries bin through deterministic encryption with ψ and produces eb:

$$eb \leftarrow \mathsf{Encrypt}_{\mathsf{DET}}(\psi, \mathsf{bin}) \tag{5.23}$$

Then, each participating *developer* uses a secure hash function $\mathsf{H}(\cdot)$ to compute the digests h_{ψ}, h_{eb}, h_{src} and h_{bin}:

$$h_{\psi} \leftarrow \mathsf{H}(\psi) \qquad h_{eb} \leftarrow \mathsf{H}(eb) \tag{5.24}$$

$$h_{\mathsf{src}} \leftarrow \mathsf{H}(\mathsf{src}) \qquad h_{\mathsf{bin}} \leftarrow \mathsf{H}(\mathsf{bin}) \tag{5.25}$$

Each *developer* in D encrypts the deterministic encryption key ψ through MA-CP-ABE encryption by using the *multi-authority policy* $\mathbb{A}$ received by *admins*, and computes its digest $h_{e\psi_d}$ with a secure hash function $\mathsf{H}(\cdot)$:

$$e\psi_d \leftarrow \mathsf{Encrypt}_{\mathsf{ABE}}(\psi, \mathbb{A}, \{pk_r\}_{r \in \mathcal{R}})\ \forall d \in D \tag{5.26}$$

$$h_{e\psi_d} \leftarrow \mathsf{H}(e\psi_d)\ \forall d \in D \tag{5.27}$$

One designated *developer* determines the timestamp t of the current update and computes the *update validation metadata* uvm and *binaries metadata* bm:

$$\mathsf{bm} := \langle h_{\mathsf{bin}}, h_{eb}, h_{\psi}, \mathsf{DKMA}, t, \langle v, \Delta t, \sigma_{\Delta tv} \rangle \rangle \tag{5.28}$$

$$\mathsf{uvm} := \langle h_{\mathsf{src}}, \mathsf{bm} \rangle \tag{5.29}$$

The developer sends both of them to all other participating *developers*. Each developer verifies their correctness by:

- recomputing h_{src} and h_{bin}, and by verifying that they match the corresponding values in uvm and bm;
- verifying signature $\sigma_{\Delta tv}$;
- verifying that digests h_{ψ} and h_{eb} are equal to the digests computed in Equation 5.24;
- verifying that DKMA is valid, as defined in Section 5.5.2;
- verifying that t is a timestamp indicating a plausible time of creation of tuples bm and uvm.

Developers in D multi-sign uvm, producing AUVM:

$$\mathsf{AUVM} = \langle \mathsf{uvm}, \sigma_{\mathsf{AUVM}} \rangle \tag{5.30}$$

Finally, participating *developers* gather the digests $h_{e\psi_d}$ and multi-sign the tuple $\langle \mathsf{bm}, \{h_{e\psi_d}\} \rangle$, producing ABM:

$$\mathsf{ABM} = \left\langle \left\langle \mathsf{bm}, \{h_{e\psi_d}\} \right\rangle, \sigma_{\mathsf{ABM}} \right\rangle \tag{5.31}$$

We highlight that the digests h_{src}, h_{bin}, h_{eb} and h_{ψ} along with signatures σ_{AUVM} and σ_{ABM} are used to guarantee integrity and authenticity of source code, binaries and related cryptographic material to *validators* and *clients*, respectively.

5.5.5 Update validation

We model the update validation procedure as $\mathsf{ValidateUpdate}(\mathsf{src}, \psi, \mathsf{AUVM}, \mathsf{ABM}, \mathsf{location})$, where src is the update source, ψ is the deterministic encryption key, AUVM and ABM are authentication material, $\mathsf{location}$ is the address of encrypted binaries. The goal of this procedure is validate source-to-binary correspondence and append ABM and $\mathsf{location}$ to the *update* and *location* layers of the multi-layer skipchain, respectively.

A designated *developer* sends AUVM, ABM and $\langle \mathsf{src}, \psi \rangle$ to *validators* over a confidential and authenticated channel. Each *validator* obtains the *admins* public keys $\{pk_a\}$ and the latest update version value v' from the *admins* and *update* skipchain layers, and verifies that AUVM, ABM, the tuple $\langle \mathsf{src}, \psi \rangle$ and version value v' are correct and authentic information by executing Algorithm 1. If all checks pass, *validators* multi-sign ABM producing σ_{VABM}:

$$\mathsf{VABM} := \langle \mathsf{ABM}, \sigma_{\mathsf{VABM}} \rangle \tag{5.32}$$

Algorithm 1 Metadata validation

function VALIDATE($\{pk_a\}$, AUVM, ABM, src, ψ, v')
  uvm $\leftarrow$ AUVM.uvm
  bm $\leftarrow$ uvm.bm
  DKMA $\leftarrow$ bm.DMKA
  $\{pk_d\} \leftarrow$ DKMA.$\{pk_d\}$
  Verify($\{pk_d\}$, AUVM.σ_{AUVM}, AUVM)
  Verify($\{pk_d\}$, ABM.σ_{ABM}, $\langle$bm, ABM.$\{h_{e\psi_d}\}\rangle$)
  Verify($\{pk_a\}$, bm.$\sigma_{\Delta tv}$, $\langle$bm.v, bm.$\Delta t\rangle$)
  Verify($\{pk_a\}$, DKMA.σ_{DKMA}, DKMA)
  $v' \overset{?}{<}$ bm.v
  AUVM.$h_{\mathsf{src}} \overset{?}{=}$ H(src)
  bin $\leftarrow$ DeterministicBuild(src)
  bm.$h_{\mathsf{bin}} \overset{?}{=}$ H(bin)
  bm.$h_{\psi} \overset{?}{=}$ H(ψ)
  $eb \leftarrow$ Encrypt$_{\mathsf{DET}}(\psi, \mathsf{bin})$
  bm.$h_{eb} \overset{?}{=}$ H(eb)

Then, a designated *validator* sends VAUM to *admins*.

Admins multi-sign the location of the update at version ABM.v producing *Authenticated Location* AL which we define as follows:

$$\mathsf{location} := \{loc_{eb}, loc_{e\psi_1}, \ldots, loc_{e\psi_{|D|}}\} \tag{5.33}$$

$$\mathsf{AL} := \langle\langle \mathsf{location}, \langle \mathsf{ABM.bm}.v, v_c\rangle\rangle, \sigma_{\mathsf{AL}}\rangle \tag{5.34}$$

where loc_{eb} is the location of encrypted updates eb, $loc_{e\psi_1}, \ldots, loc_{e\psi_{|D|}}$ are the locations of encrypted keys $e\psi_d$ and v_c is a unique counter value used to denote the number of location changes for the same version v which, in this procedure, is initialized to zero. Value v_c is increased in case of updates to the location and controlled by the *witnesses* accordingly. A designated *admin* starts the PBFT-CoSi protocol with *witnesses* by sending VAUM and AL to the *witness* leader. During the *pre-prepare* phase of the PBFT-CoSi protocol, *witnesses* verify multi-signatures σ_{VAUM} and σ_{AL} with the latest *validators* and *admins* public keys specified in the *validators* and *admins* skipchain layers, respectively. If verification succeeds, *witnesses* append VAUM and AL to the *update* and *location* layers, respectively. At the end of PBFT-CoSi *commit* phase, the *witnesses* leader returns to the designated *admin* an attestation of the correct execution of the protocol.

5.5.6 Client key refresh

We model the client key refresh procedure as ClientKeyRefresh($cid, P_{cid}, \mathcal{A}$), where *cid* is the unique identifier of the client, P_{cid} is the set of *original attributes* associated to the client by the *software house* and $\mathcal{A}$ is the latest version of the

ABE attributes matrix (Section 5.5.3). The procedure generates a set of *ABE keys* SK_{cid} assigned to the client to decrypt update binaries (Section 5.5.8).

A *client* first authenticates to the *authentication server* by presenting appropriate credentials that include the client identifier *cid*. If authentication is successful, the *authentication server* retrieves the *client original attributes* P_{cid} from its own database. By using the *ABE attributes matrix* $\mathcal{A}$, the *authentication server* assigns the set of client ABE attributes $\mathcal{C}$ depending on the *original attributes* P_{cid} associated to the client.

For ease of presentation, we model P_{cid} as a set of indexes to the enumerated set of *original attributes*, as described in the *update policies* procedure (Section 5.5.3), that is: $P_{cid} \subseteq [|P|]$.

Let us consider a key reliability parameter $f_k \in [0, n_r - k_r]$ that regulates the *clients* tolerance to *ABE servers* key rotations. As an example, a value $f_k = 1$ guarantees that even if one *ABE server* rotates his keys, the *client* is still able to decrypt future updates. The *authentication server* chooses a subset of ABE servers $\bar{\mathcal{R}} \subseteq \mathcal{R}$ for which he releases decryption keys, such that $|\bar{\mathcal{R}}| = (f_k + k_r + 1)$.

The matrix of client *ABE attributes* $\mathcal{C}$ is computed as:

$$\mathcal{C} = \left(\alpha^j_{\zeta(r)} : \alpha^j_{\zeta(r)} \in \mathcal{A}\right), \forall r \in \bar{\mathcal{R}}, \forall j \in P_{cid} \tag{5.35}$$

For ease of presentation, we denote as $\mathcal{C}_r$ the row of $\mathcal{C}$ associated to server r. The *authentication server* sends to each *ABE server* $r \in \bar{\mathcal{R}}$ the row $\mathcal{C}_r$ over a secure channel. Each server r computes a set of *ABE keys* $SK_{cid,r}$ as:

$$SK_{cid,r} := \{sk : sk \leftarrow \mathsf{KeyGen}(cid,\ sk_r,\ e_r), \forall e_r \in \mathcal{C}_r\} \tag{5.36}$$

where we recall that sk_r is server's r secret key. Each *ABE server* $r \in \bar{\mathcal{R}}$ sends the set of keys $SK_{cid,r}$ to the *client* through the *authentication server*. Once all *ABE servers* in $\bar{\mathcal{R}}$ have responded, the *client* can compute the matrix $SK_{cid} = (SK_{cid,r}), \forall r \in \bar{\mathcal{R}}$.

The *authentication server* can adopt multiple strategies to establish the value f_k and the servers of the set $\bar{\mathcal{R}}$. The number of keys the *client* receives depends on the value f_k. The value f_k must lie in the range $[0, n_r - k_r - 1]$ and $|\bar{\mathcal{R}}| = (f_k + k_r + 1)$ because, if $f_k = 0$ then $|\bar{\mathcal{R}}| = k_r + 1$ and thus the *client* is able to satisfy policy $\mathbb{A}$ because *ABE servers* issue the minimum amount of $k_r + 1$ keys required to satisfy an original attribute (see Section 5.5.3). Moreover, f_k must not exceed $n_r - k_r - 1$ because $|\bar{\mathcal{R}}| \leq n_r$. To one extreme, choosing $f_k = 0$ minimizes the number of keys sent and managed by the *client*, however it forces to refresh his keys when any of the keys belonging to *ABE servers* in $\bar{\mathcal{R}}$ is rotated. On the other extreme, choosing $f_k = n_r - k_r - 1$ maximizes the number of keys sent to the *client*, but the *client* is forced to refresh his keys only when $n_r - k_r$ *ABE servers* keys have been rotated. As long as $f_k < n_r - k_r - 1$, the *authentication server* can choose which *ABE servers* issue the new keys. This may be useful to load balance the key generation process among *ABE servers* in case of bursty key refresh workloads.

5.5.7 Check for updates

The goal of this procedure, which is started by a *client*, is to efficiently update the *client's* copy of the multi-layer skipchain so that the *client* can determine whether new software updates are available. We model the *check for updates* procedure as $\mathsf{CheckForUpdates}(\tau_t)$, where τ_t is the last authenticated timestamp belonging to the *time* skipchain layer that is known by the *client* and t denotes the t^{th} execution of *check for updates*. The procedure returns the skipblocks between the current latest skipblock of each layer of the multi-layer skipchain maintained by *witnesses*, and the skipblocks pointed by τ_t. Moreover, it returns the latest authenticated timestamp that the *client* uses in the next invocation of *check for updates*.

The *client* requests the latest timestamp τ'_{t+1} to at least k_w *witnesses* over an authenticated channel. If all timestamps are equal, then the *client* sends $\langle \tau_t, \tau'_{t+1} \rangle$ to any *witness* over an authenticated channel. The *witness* determines the *client's* last skipblock for each skipchain layer by following the hash pointer to the parent skipchain of each layer, starting from the *location* skipblock pointed by τ_t. For each layer, the *witness* determines the shortest chain of multi-signed forward pointers between the *client*'s last skipblock and the current latest skipblock. The *witness* sends to the *client* the required skipblocks of the *admins* and *witnesses* layers, the multi-signature chains and the latest skipblock of all other layers. The *client* validates the multi-signature chains by using the public keys contained in the *witnesses* skipblocks, validates the *witnesses* skipblocks with the public keys contained in the *admins* skipblocks, and finally validates the timestamp τ'_{t+1} obtained in the beginning by using the latest set of *witnesses* public keys and by checking that the timestamp respects the latest *freshness tolerance* value Δt (see Section 5.4.2). If skipchain validation succeeds, the *client* sets $\tau_{t+1} = \tau'_{t+1}$ for the following invocation of the same *check for updates* procedure. The *client* checks the latest *attributes* skipblock and executes the *client key refresh* procedure if he needs to refresh his keys (see Section 5.5.6). If a new *update* skipblock is present, then the *client* executes the *download and decrypt* procedure described in Section 5.5.8.

Moreover, we observe that the network cost of transferring the multi-signature chains to *clients* is logarithmic in the amount of skipblocks of each layer between τ_t and τ_{t+1} [96].

5.5.8 Download and decrypt

The goal of the *download and decrypt* procedure is to let the *client* download, authenticate and decrypt a new software update after he determines its availability through the *check for updates* procedure (Section 5.5.7). We model the procedure as $\mathsf{DownloadAndDecrypt}(\mathsf{VABM}, \mathsf{AL}, SK_{cid})$, where VABM and AL are authenticated data structures obtained from the *check for updates* procedure, and SK_{cid} is the *client* ABE keys obtained from the *client key refresh* procedure.

We represent the procedure in Algorithm 2. In lines 2 through 10 the *client* downloads and decrypts $e\psi_i$, which is the deterministic encryption key ψ encrypted by *developer* i with ABE encryption (Equation 5.26). If the digest of the

Algorithm 2 Download and decrypt

1: **function** DOWNLOADDECRYPT(VABM, AL, SK_{cid})
2: $i \leftarrow 0$
3: **repeat**
4: $i \leftarrow i + 1$
5: $loc_{e\psi_i} \leftarrow \mathsf{AL.location}.loc_{e\psi_i}$
6: $e\psi_i \leftarrow \mathsf{Download}(loc_{e\psi_i})$
7: $\mathsf{H}(e\psi_i) \stackrel{?}{=} \mathsf{VABM.ABM}.h_{e\psi_i}$
8: $\psi \leftarrow \mathsf{Decrypt}_{\mathsf{ABE}}(SK_{cid},\ e\psi_i)$
9: $h \leftarrow \mathsf{VABM.ABM.bm}.h_{\psi}$
10: **until** $i \stackrel{?}{=} |D| \parallel \mathsf{H}(\psi) \stackrel{?}{=} h$
11: $loc_{eb} \leftarrow \mathsf{AL.location}.loc_{eb}$
12: $eb \leftarrow \mathsf{Download}(loc_{eb})$
13: $\mathsf{H}(eb) \stackrel{?}{=} \mathsf{VABM.ABM.bm}.h_{eb}$
14: $\mathsf{bin} \leftarrow \mathsf{Decrypt}_{\mathsf{DET}}(\psi, eb)$

decrypted key is not equal to the digest included in the VABM data structure, he tries to download and decrypt the deterministic key encrypted by developer $i + 1$ for all possible developers. When the *client* finds a valid key, he can start the decryption procedure of the update binaries. To this aim, he downloads the encrypted binaries from the *distribution infrastructure* (line 12), validates their authenticity (line 13) and decrypts them (line 14). We recall that the authenticity of digest h_{eb} is guaranteed by σ_{VABM} included in VABM and validated during the *check for updates* procedure. Finally, the *client* can install the update binaries bin and complete the update procedure.

5.5.9 Key management

The *key management* operations are accomplished by *admins*, who collectively act as a root certification authority, and thus are the only role responsible for rotating the public keys of other roles. Each key management operation produces a new authenticated set of public keys for a specific role. With the only exception of the *developers* role, the new set is sent to *witnesses*, who verify its authenticity, validate its correctness and append it to the corresponding *keys metadata* skipchain layer. In the following we describe key management operations for the different roles: *admins*, *witnesses*, *validators*, *ABE servers* and *developers*.

Admins. *Admins* manage their keys $\{pk_a\}$, authenticated in attestation AKMA, by self-authenticating a new set of public keys $\{pk_a\}'_{a\in[n_a]}$ in a new attestation AKMA′. *Admins* must specify the new validity period $\langle \mathsf{AKMA}'.v_b, \mathsf{AKMA}'.v_e \rangle$, which must not overlap with the validity period $\langle \mathsf{AKMA}.v_b, \mathsf{AKMA}.v_e \rangle$ (Equation 5.5). *Admins* may also change the multi-signature threshold by specifying $\mathsf{AKMA}'.k_a \neq \mathsf{AKMA}.k_a$. A threshold of $\mathsf{AKMA}.k_a + 1$ *admins* multi-sign AKMA′ and send it to *witnesses*, that append it only if it satisfies the security constraints (non-overlapping validity periods, threshold and validity of signing keys).

Algorithm 3 Rotate ABE servers

1: **function** RotateABEServers($\{sk_a\}$, $\mathcal{R}_c$, $\mathcal{R}'$, v)
2: $\mathcal{R} = (\mathcal{R} \setminus \mathcal{R}_c) \cup \mathcal{R}'$
3: $\langle sk_r, pk_r \rangle \leftarrow \mathsf{AuthSetup}(r)\ \forall r \in \mathcal{R}'$
4: $\sigma_{\mathsf{RKMA}'} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \{pk_r\}_{r \in \mathcal{R}}, k_r, v'_b, v'_e \rangle)$
5: $\mathsf{RKMA}' = \langle \langle \{pk_r\}_{r \in \mathcal{R}}, k_r, v'_b, v'_e \rangle, k_a, \sigma_{\mathsf{RKMA}'} \rangle$
6: $\mathcal{A}' \leftarrow \mathsf{UpdatePolicy}(\mathbb{P}, v, k_r, \mathcal{R})$
7: $\sigma_{\mathsf{AAM}'} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \mathcal{A}', v \rangle)$
8: $\mathsf{AAM}' = \langle \langle \mathcal{A}', v \rangle, \sigma_{\mathsf{AAM}'} \rangle$
9: $\mathsf{Append}(\mathsf{L_r}, \mathsf{RKMA}')$
10: $\mathsf{Append}(\mathsf{L_{at}}, \mathsf{AAM}')$

Witnesses, validators. *Admins* manage the public keys of *witnesses* and *validators*, authenticated in a key metadata attestation KMA (specifically, WKMA, VKMA for the two roles, see Section 5.5.2), by computing a new attestation KMA′. *Admins* must specify the new validity period $\langle \mathsf{KMA}'.v_b, \mathsf{KMA}'.v_e \rangle$ which must not overlap with the validity period $\langle \mathsf{KMA}.v_b, \mathsf{KMA}.v_e \rangle$. A threshold of $\mathsf{KMA}.k_a + 1$ *admins* multi-sign the new KMA′ and send it to *witnesses*, that append it only if it satisfies the due security constraints.

ABE servers. *Admins* manage the public keys of *ABE servers*, authenticated in a key metadata attestation RKMA (see Section 5.5.2), by computing a new attestation RKMA′ as outlined in Algorithm 3. If *admins* rotate *ABE servers* public keys, *admins* must also execute the *attribute derivation* phase of the *update policies* procedure to update the *authenticated attributes matrix* $\mathcal{A}$ (Section 5.5.3). If *admins* change the value of k_r to k'_r after removing or adding new *ABE servers*, *admins* must also execute the *policy translation* phase of the *update policies* procedure to compute a new *multi-authority policy* $\mathbb{A}$ that complies to the new k'_r. *Admins* can rotate *ABE servers* keys to promptly recover from the compromise of up to k_r *ABE servers* without executing $\mathsf{Setup}(1^\lambda)$ again, by executing Algorithm 3. In line 2 *admins* update the set of *ABE servers* by excluding the set of compromised *ABE servers* $\mathcal{R}_c \subset \mathcal{R}$ and including the set of new *ABE servers* $\mathcal{R}'$ (where $|\mathcal{R}_c| = |\mathcal{R}'|$). In line 3 each new *ABE server* in $\mathcal{R}'$ generates his signing key pair. In lines 4 and 5 *admins* authenticate the new set of *ABE servers* public keys. In lines 6 through 8 *admins* update and authenticate the new attribute matrix $\mathcal{A}'$ in a new attestation AAM′ and finally in lines 9 and 10 *admins* append the new authenticated data structures AAM′ and RKMA′ to the appropriate skipchain layers.

Developers. *Admins* authenticate *developers* keys with the DKMA attestation at every execution of the *update authentication* procedure (see Equation 5.20). For this reason, *developers* can freely rotate keys between software releases, and no skipchain layer is dedicated to tracking their evolution.

We highlight that key management operations that alter the number of actors within a specific role might impact the security and the performance of the system. Removing an actor might require the remaining actors within the same

role to guarantee higher level of availability or to operate higher workloads to comply to the required multi-signatures threshold. At the same time, lowering the threshold would guarantee the same level of performance but decrease the security level of the system. As an example, reducing the number *validators* without also reducing the corresponding multi-signature threshold $\mathsf{VKMA}.k_v$ implies that a higher percentage of *validators* must be available during the *update validation* procedure. However, reducing $\mathsf{VKMA}.k_v$ implies tolerating a lower amount of malicious *validators*. We note that the number of *witnesses* is $n_w = 3k_w + 1$ due to the use of PBFT protocol, therefore *witnesses* must not be less than 4.

5.6 Security analysis

Survivability is guaranteed by the adoption of multi-signatures coupled with validity thresholds and by the accurate definition of access control policies, as described in Section 5.5.3. This design choice allows the compromise of up to a threshold of actors (admins, developers, ABE servers, witnesses, verifiers) still guaranteeing that an adversary cannot forge any authenticated cryptographic material produced in the procedures of our proposal. The key rotation procedure (Section 5.5.9) ensures *recoverability*. In fact, if a threshold of admins, witnesses, verifiers, ABE servers or developers keys are compromised, then admins can recover the system to a safe state by rotating the compromised keys as soon as the incident is detected.

Authenticity of software updates is protected by admins and developers multi-signatures, who digitally sign software updates data and metadata during the update authentication procedure (Section 5.5.4). The authenticity of developers and admins digital signatures is in turn guaranteed by the multi-layer skipchain that allows admins to manage keys and act as a certification authority.

An adversary can try to break authenticity in several ways: by compromising admins to issue rogue keys, by compromising developers to authenticate malicious source code, by compromising validators to approve malicious update binaries or witnesses to equivocate or fork the multi-layer skipchain. However, *authenticity* does not suffer from single points of failure as the validity of a role's multi-signature is determined by an admin-defined threshold on the number of signers, which is authenticated through attestations published on the multi-layer skipchain (Section 5.5.2). As a result, an adversary is unable to break authenticity of software updates because we assume it is not able to compromise more than a threshold of actors for each role.

Confidentiality. We evaluate confidentiality guarantees by distinguishing the confidentiality of software updates binaries and of source code. The former is protected against an adversary who intercepts the binaries, either by compromising the distribution infrastructure or by intercepting them while being sent to and downloaded from the distribution infrastructure. The adversary can try to break the confidentiality of intercepted software update binaries by violating ABE servers to recover the decryption key ψ. As a result, the confidentiality of software update binaries does not suffer from single points of failure because, as described in Section 5.5.3, the adversary must violate at least $k_r + 1$ ABE

servers to be able to obtain the decryption key.

The confidentiality of software update source code could be violated by corrupting a developer or a validator during the validation phase. Concerning this issue, we consider the typical approach of the literature assuming that the confidentiality of software updates source code is based on weakest-link security. Attacks by one developer could be even minimized by adopting software management techniques that segment source code and prevent one developer to access the whole code and/or by detailed logging and forensics mechanisms, but the integration of similar solutions is out of the scope of this proposal.

Freshness and timeliness. The multi-layer skipchain ensures *freshness* of software updates. The adoption of PBFT along with non-equivocation mechanisms guarantees to clients a consistent view of the skipchain state and, more specifically, of its latest skipblocks. As described in Section 5.5.7, non-equivocation is obtained by querying at least $k_w + 1$ witnesses so that at least an honest witness is queried. The honest witness guarantees to detect equivocation attacks if he returns a response which is inconsistent with the responses of other possible malicious witnesses. Moreover, the *time* skipchain layer allows clients to detect freeze attacks by an adversary who controls the communication channel of the client and presents a stale view of the skipchain.

The *timely* and scalable distribution of software updates is made possible by the design of the software update encryption mechanism that produces a single ABE ciphertext for all clients, thus allowing us to leverage existing distribution infrastructures, such as Content Delivery Networks.

Transparency is guaranteed by validators through the update validation procedure (Section 5.5.5). Assuming that validators use a trusted compiler, they can detect attacks against source-to-binary correspondence in which an adversary induces developers into signing backdoored software update binaries that do not correspond to the original update source code.

5.7 Costs analysis and performance evaluation

We evaluate the overhead introduced by the proposed framework and demonstrate that it achieves practical performance. First, we analyze the computational, network and storage costs of our contributions: MA-CP-ABE extension and distributed update authentication protocol. Then, we evaluate the framework performance by considering timings of actual state-of-the-art cryptographic libraries at multiple security levels and scenarios of increasing complexity.

5.7.1 Costs analysis

We analyze the computational costs of MA-CP-ABE in terms of relevant primitive cryptographic operations and complexity of access control policy formulas by referring to Table 5.2. We denote the pairing operation as e, exponentiation in $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ as p_1, p_2 and p_T respectively, and hash to $\mathbb{G}_2$ operation as H. Moreover, we denote as x the number of rows of the linear secret sharing scheme (LSSS) matrix used by the ABE scheme, that is equal to the number of logic gates in the translated access control policy $\mathbb{A}$ plus one [78]. As we discussed

Operation	Role	Procedure	Costs
ABE encryption	Developer	(1)	$p_T + x\,(H + 3\,p_1 + p_2 + 2\,p_T)$
ABE decryption	Client	(6)	$x\,(3\,e + H + p_T)$
ABE keygen	ABE server	(4)	$a(2\,H + p_1 + 3\,p_2)$

(1) Auth. update, (4) Key refresh, (6) Download & decrypt

Table 5.2: Computational costs

in Section 5.5.3, the number of logical gates in $\mathbb{A}$ can be computed by knowing the number of gates in the original access control policy $\mathbb{P}$, that we denote as γ, the number of ABE servers n_r, and the ABE server threshold k_r. The value of x can be computed as following:

$$x = \gamma + (\gamma + 1) \cdot [(k_r + 1) \cdot \binom{n_r}{(k_r + 1)} - 1] + 1 \tag{5.37}$$

We observe that the encryption and decryption costs are linear in x, which increases as a binomial function of n_r and k_r. We recall that the value n_r represents the number of heterogeneous ABE servers that do not share common-mode failures [13]. Thus, it is unlikely that n_r exceeds a few units. We also observe that decryption is typically more expensive than encryption due to the three pairing operations $(3 \cdot e)$ and to the possibility of using optimizations for fixed bases in point scalar multiplication operations in the encryption operation [101] (only the point scalar multiplication p_2 is computed on a variable base). Finally, the key generation phase depends on the amount of *original attributes* granted to the client, that we denote as a (see Section 5.5.6). This represent the worst case of a client requesting keys for all attributes, such as at setup time. In other procedures, such as policy updates (see Section 5.5.3), the client may request keys for a subset of attributes. If some load balancing strategy is applied, then the procedure in each key generation would involve just a subset of the ABE servers (see Section 5.5.6).

We discuss network and storage overhead introduced by ABE cryptographic material by referring to Table 5.3. The first column (*data*) describes the type of cryptographic material. The second and third columns (*role* and *procedure*) describe which actors are affected by these costs and in which procedures of the framework, respectively. The fourth column (*type*) indicates whether the material affects storage or network costs for each actor. The last column includes the costs with regard to the type of material, where we denote the size of an element belonging to the groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ as G_1, G_2 and G_T, the number of developers that participate in the update authentication procedure as $|D|$, the key reliability parameter as f_k, and the number of original attributes granted to a client as a. The size of the ABE ciphertext grows linearly as a function of x, thus increasing as a binomial function of n_r and k_r for ABE encryption

Data	Role	Proc.	Type	Costs
ABE ciphertext	Developer	(1)	$\odot$	$G_T + x\,(2\,G_1 + G_2 + G_T)$
	Client	(6)	$\odot$	
	Dist.Inf	(3)	$\odot\otimes$	$\lvert D\rvert\,(G_T + x\,(2\,G_1 + G_2 + G_T))$
ABE client keys	ABE server	(4)	$\odot$	$a\,(G_1 + G_2)$
	Client	(4)	$\odot\otimes$	$a\,(f_k + k_r + 1)\,(G_1 + G_2)$

(1) Auth. update, (3) Publish, (4) Key refresh, (6) Download & decrypt
$\odot$: network costs, $\otimes$: storage costs

Table 5.3: Storage and network costs

and decryption costs. Moreover, the size of the keys received and maintained by each client depends on the values a, k_r and f_k because each of $(f_k + k_r + 1)$ ABE servers send a keys of size $(G_1 + G_2)$ (see Section 5.5.6).

As described in Section 5.5.4, our distributed update authentication protocol includes the costs due our MA-CP-ABE extension, and costs due to distributed key agreement and deterministic encryption. We observe that any deterministic symmetric encryption scheme adds only a minor computational overhead with respect to probabilistic symmetric encryption schemes [64], that in turn are negligible with regard to asymmetric encryption schemes. Moreover, they do not add relevant network overhead. Thus, any role that is able to operate MA-CP-ABE is also able to support deterministic encryption. We analyze the costs of the distributed key agreement by considering an instantiation based on authenticated group key agreement protocol [34]. Each developer executes $2|D| + 1$ group exponentiations and $3|D|$ signatures, where $k_d < |D| \leq n_d$ is the number of developers that participate in the key agreement protocol. Moreover, the network costs of each developer consist of $3|D|$ group elements and $3|D|$ signatures. If we consider realistic values of $|D|$ being within the tens of developers, the amount of group elements and digital signatures that a developer must compute and transmit introduce feasible computational and network costs.

5.7.2 Performance evaluation

The performance evaluation of our MA-CP-ABE extension is based on two pairing-friendly curves: BN256 [95] and BN462 [106], which account for security levels of about 100 and 128 bits. In Table 5.4 we report timings expressed in clock cycles and group element sizes expressed in bits. We obtained timings by using the implementations included in the MCL library v1.10 [93] compiled for Intel i7-8665U processor. Moreover, we analytically computed element sizes by using the curves parameters. For BN256 $G_1 = 256$ and $G_2 = G_T = k \cdot G_1 = 3072$ bits, and for BN462 $G_1 = 462$ and $G_2 = G_T = k \cdot G_1 = 5544$ bits, where G_1 is computed by the designers of the curve with regard to the security level and $k = 12$ is the embedding degree of both curves. The sizes are computed by

Curve	λ	Timings [Clock cycles]					Size [bit]		
		p_1	p_2	p_T	e	H	G_1	G_2	G_T
BN256	100	97 k	210 k	332 k	638 k	131 k	256	3063	3063
BN462	128	719 k	1.6 M	1.9 M	4.8 M	788 k	462	5535	5535

Table 5.4: Curves parameters and performance

n_r	k_r	γ	x	BN256			BN462		
				Enc	Dec	Size	Enc	Dec	Size
2	1	1	4	1.2 ms	2.0 ms	3.7 kB	7.3 ms	14 ms	6.7 kB
2	1	2	6	1.7 ms	3.0 ms	5.4 kB	11 ms	21 ms	9.7 kB
2	1	5	12	3.3 ms	5.9 ms	10.4 kB	21 ms	43 ms	18.7 kB
2	1	10	22	6.0 ms	11 ms	18.7 kB	38 ms	78 ms	33.7 kB
2	1	50	102	28 ms	51 ms	85.2 kB	177 ms	363 ms	153.8 kB
3	1	1	12	3.3 ms	5.9 ms	10.4 kB	21 ms	43 ms	18.7 kB
3	1	2	18	4.9 ms	8.9 ms	15.4 kB	32 ms	64 ms	27.7 kB
3	1	3	24	6.6 ms	12 ms	20.4 kB	42 ms	85 ms	36.7 kB
3	1	10	66	18 ms	33 ms	55.3 kB	115 ms	235 ms	99.8 kB
3	1	50	306	83 ms	152 ms	255.0 kB	530 ms	1.1 s	460.2 kB
4	1	10	132	36 ms	65 ms	110.2 kB	229 ms	470 ms	198.9 kB
5	2	3	120	33 ms	59 ms	100.2 kB	208 ms	427 ms	180.9 kB

Table 5.5: Evaluation of encryption and decryption times, and of the ciphertext size

considering compressed elliptic curve coordinates and uncompressed finite field elements. In such a way, the size of G_1 and G_2 is equal to the size of an element of the field over which the curve is defined.

To estimate the performance of the approach in realistic scenarios, in Tables 5.5 and 5.6 we propose results based on a set of parameters that are representative for real-world scenarios. In both tables, we compute timings from the clock cycles reported in Table 5.4 and formulas reported in Tables 5.2 and 5.3, and in Equation (5.37) by considering a modern x86_64 CPU operating at $4.8GHz$. The considered parameters influence the system performance: n_r, k_r, γ, a and f_k. In Table 5.5 we show encryption and decryption times as well as ciphertext size which depend on parameters n_r, k_r and γ. Moreover, in Table 5.6 we report key generation times and the sizes of decryption keys for ABE servers and clients, which depend on parameters a, k_r and f_k.

Results in Table 5.5 highlight that decryption, run by clients, is the most expensive operation and typically costs twice the encryption, run by developers. Our proposal is practical in realistic scenarios where the number of ABE servers n_r and the amount of tolerable malicious servers k_r is of few units. For example, when $n_r = 3$, $k_r = 1$ and γ is within tens of logical gates, timings are acceptable (if $\gamma \in [1, 10]$ then decryption takes between 60ms to 200ms). A possible instance of the original access policy with $\gamma = 3$ is $\mathbb{P} = \mathsf{A} \wedge (\mathsf{D} \vee (\mathsf{B} \wedge \mathsf{C}))$.

a	$(f_k + k_r + 1)$	BN256			BN462		
		Keygen	Server	Client	Keygen	Server	Client
2	2	413 us	6.7 kB	13.3 kB	3.0 ms	12.0 kB	24.0 kB
	3			20.0 kB			36.0 kB
	6			39.9 kB			72.1 kB
3	3	619 us	10.0 kB	30.0 kB	4.5 ms	18.0 kB	54.1 kB
30	4	6.2 ms	99.8 kB	399.4 kB	45 ms	180.2 kB	720.7 kB

Table 5.6: Evaluation of key generation timings and key sizes

Table 5.5 also reports the space overhead of a single ABE ciphertext in columns "Size", which corresponds to the network cost for clients during the download and decrypt procedure, and for developers during the authenticate update procedure (see Table 5.3). As highlighted in Table 5.3, this value must be multiplied by $|D|$ so to compute the network and storage costs of the distribution infrastructure bears during the publish procedure. In realistic scenarios where the number of ABE servers n_r and the amount of tolerable malicious ABE servers k_r are within a few units, and the number of logical gates γ of the original access policy is within tens of gates, the overhead does not exceed 200KB. When software updates are in the order of hundreds of kilobytes, the space overhead of one ABE ciphertext has a size comparable to that of the update. This overhead becomes negligible in scenarios where software updates tend to be in the order of ten megabytes or more.

Our performance evaluation shows that the timings for generating ABE keys are acceptable even in cases where a client satisfies several tens of attributes, as evidenced in Table 5.6. Curve BN256 allows to compute a single ABE key in 206.5 microseconds, allowing to generate 4842 ABE keys per second. Curve BN462 allows to compute a single ABE key in 1.5 milliseconds, allowing to generate 666 ABE keys per second. We note that these throughput values consider the maximum achievable throughput of a single machine with a single core executing the key generation procedure. Higher throughput values can be obtained by horizontally and vertically scaling ABE servers or by choosing a key reliability value f_k such that $0 \leq f_k \leq n_r - k_r - 1$, so that the authentication server can apply load balancing strategies to share the load between ABE servers, as we discussed in Section 5.5.6. The network costs for ABE servers and the network and storage costs for clients in scenarios where a client satisfies a few attributes, tend to be several tens of kilobytes. In extreme scenarios where a client satisfies several tens of attributes of an unrealistically complex access policy, the space overhead of ABE keys is several hundreds of kilobytes.

Our analysis shows that the framework is practical for a number of ABE servers in the order of several units, and a number of developers and logical gates in access control policies in the order of tens. In particular, the proposed framework is better suited for scenarios in which software updates tend to be in the order of megabytes. As a requirement to support the framework, clients must have enough memory and storage capacity to maintain ABE decryption

keys and decrypt ciphertexts, that are typically of tens of kilobytes each.

5.8 Final remarks

We propose an original framework that allows the secure and survivable distribution of confidential software updates. This framework is based on multi-authority attribute-based encryption, and extends its key generation procedure with an original technique to guarantee survivability. It is based on a distributed infrastructure with no single points of failure which is able to guarantee availability and security even in the presence of partial compromises. We demonstrate the practicality of the proposal through a performance evaluation of our original key generation technique and of the encryption scheme in the context of secure software updates. The results show that the proposed framework can achieve practical performance at 128-bit security level on modern computers in realistic settings. This framework paves the way to the design of secure and robust business-oriented architectures for the distribution of confidential software updates. Our proposal highlights even some interesting open problems, such as the protection of source code confidentiality with no weakest link security assumption, and the possibility of enabling software transparency without relying on third parties. These issues may be addressed in future work that may well be integrated into the proposed framework.

Chapter 6

Conclusions

Recent incidents show that adversaries are becoming increasingly sophisticated, to the extent that they are now capable of compromising even the trusted components of existing IT systems. These events highlight that, to thwart sophisticated adversaries, IT systems must be able to survive successful attacks. To address this urgent challenge, in this thesis we consider the most critical IT systems and redesign them to be *survivable*. We introduce the first survivable zero trust architecture, address limitations of existing password-based survivable SSO protocols, propose the first passwordless survivable SSO protocol and present the first survivable software update framework. The contributions of this thesis advance the field of survivability and intrusion tolerance which is still open to several improvements. New research directions could investigate survivability of IT systems that are going to be essential for future society and that have not been considered in this thesis, improve usability of existing proposals or address the technological and organizational challenges that need to be solved to ease the adoption of survivable systems.

Bibliography

[1] FIDO Alliance Specifications Overview. `https://fidoalliance.org/specifications/`.

[2] eXtensible Access Control Markup Language (XACML) Version 3.0. 22. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`, 2013.

[3] Regulation (eu) no 910/2014 of the european parliament and of the council of 23 july 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/ec. Technical report, European Parliament, Council of the European Union, 2014.

[4] Detecting abuse of authentication mechanisms. Technical Report PP-20-1485, National Security Agency, Dec. 2020.

[5] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. PASTA: PASsword-based Threshold Authentication. In *Proc. ACM SIGSAC Conf. Computer and Communications Security*, 2018.

[6] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A Practical System for Binary Transparency. In Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 11025. Springer, 2018.

[7] FIDO Alliance. Client to Authenticator Protocol (CTAP) v2.1, Jun. 2021. `https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html`.

[8] Moreno Ambrosin, Christoph Busold, Mauro Conti, Ahmad-Reza Sadeghi, and Matthias Schunter. Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution Over Untrusted Cache-enabled Networks. In *Computer Security - ESORICS*, 2014.

[9] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable and Secure Computing*, 2010.

[10] Mauro Andreolini, Marcello Pietri, Stefania Tosi, and Andrea Balboni. Monitoring large cloud-based systems. In *4th Int'l Conf. Cloud Computing and Services Science, CLOSER*. SciTePress, 2014.

[11] Giovanni Apruzzese, Fabio Pierazzi, Michele Colajanni, and Mirco Marchetti. Detection and Threat Prioritization of Pivoting Attacks in Large Networks. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[12] Claudio Agostino Ardagna, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Eros Pedrini, and Pierangela Samarati. An XACML-Based Privacy-Centered Access Control System. In *Proc. First ACM Workshop on Information Security Governance*, 2009.

[13] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 2004.

[14] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 1995.

[15] Amy Babay, Thomas Tantillo, Trevor Aron, Marco Platania, and Yair Amir. Network-Attack-Resilient Intrusion-Tolerant SCADA for the Power Grid. In *Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks*, 2018.

[16] Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable Security Analysis of FIDO2. In *Annual Int'l Cryptology Conf.*, 2021.

[17] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server. *5th IEEE European Symp. Security and Privacy*, 2019.

[18] Amos Beimel. Secure Schemes for Secret Sharing and Key Distribution. PhD thesis, Technion-Israel Institute of Technology, 1996.

[19] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated Key Exchange Secure Against Dictionary Attacks. In *Int'l Conf. Theory and Applications of Cryptographic Techniques*. Springer, 2000.

[20] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Annual Int'l Cryptology Conf.*, 1993.

[21] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *HotSec*, 2006.

[22] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. *ACM SIGMOD Record*, 1995.

[23] Philip A Bernstein and Nathan Goodman. Serializability Theory for Replicated Databases. *Jour. Computer and System Sciences*, 1985.

[24] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *44th IEEE/IFIP Int'l Conf. Dependable Systems and Networks*, 2014.

[25] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-based Encryption. In *Proc. IEEE Symp. Security and Privacy*, 2007.

[26] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *European Symp. Security and Privacy*. IEEE, 2016.

[27] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. The memory-hard Argon2 password hash and proof-of-work function. Internet-Draft draft-irtf-cfrg-argon2-13, Internet Engineering Task Force, Mar. 2021. Work in Progress.

[28] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and Dave Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, IETF, 2008.

[29] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *Int'l Workshop on Public Key Cryptography*, 2003.

[30] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *Int'l Work. Public Key Cryptography*, 2003.

[31] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. In *Proc. Int'l Conf. Theory and Application of Cryptology and Information Security*, 2001.

[32] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symp. Security and Privacy*, 2012.

[33] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for Authentication and Key Establishment*. Springer Berlin, Heidelberg, 2020.

[34] Emmanuel Bresson and Dario Catalano. Constant Round Authenticated Group Key Agreement via Distributed Computation. In *Proc. Int'l Work. Public Key Cryptography*, 2004.

[35] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proc. IEEE Symp. Security and Privacy*, 2008.

[36] Mark Campbell. Beyond Zero Trust: Trust Is a Vulnerability. *Computer*, 2020.

[37] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive Security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 1997.

[38] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. A look in the mirror: Attacks on package managers. In *Proc. 15th ACM Conf. Computer and Communications Security*, 2008.

[39] Damien Cash, Matthew Meltzer, Sean Koessel, Steven Adair, and Thomas Lancaster. Dark halo leverages solarwinds compromise to breach organizations, 2020. `https://www.volexity.com/blog/2020/12/14/dark-halo-leverages-solarwinds-compromise-to-breach-organizations/`.

[40] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[41] Jeremy Clark and Paul C Van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symp. Security and Privacy*, 2013.

[42] Ivan Damgård, Thomas P Jakobsen, Jesper Buus Nielsen, and Jakob I Pagter. Secure Key Management in the Cloud. In *IMA Int'l Conf. Cryptography and Coding*. Springer, 2013.

[43] Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In *Work. Theory and Application of Cryptographic Techniques*, 1987.

[44] Casimer DeCusatis, Piradon Liengtiraphan, Anthony Sager, and Mark Pinelli. Implementing Zero Trust Cloud Networks with Transport Access Control and First Packet Authentication. In *IEEE Int'l Conf Smart Cloud (SmartCloud)*. IEEE, 2016.

[45] Suparna Dhar and Indranil Bose. Securing IoT Devices Using Zero Trust and Blockchain. *Journal of Organizational Computing and Electronic Commerce*, 2020.

[46] Dayna Eidle, Si Ya Ni, Casimer DeCusatis, and Anthony Sager. Autonomic Security for Zero Trust Networks. In *IEEE 8th Ann. Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, 2017.

[47] Karim Eldefrawy, Rafail Ostrovsky, Sunoo Park, and Moti Yung. Proactive Secure Multiparty Computation with a Dishonest Majority. In *Int'l Conf. Security and Cryptography for Networks*. Springer, 2018.

[48] Robert J Ellison, David A Fisher, Richard C Linger, Howard F Lipson, and Thomas Longstaff. Survivable Network Systems: An Emerging Discipline. Technical report, Carnegie-mellon Univ Pittsburgh PA Software Engineering Inst, 1997.

[49] Florian M. Farke, Lennart Lorenz, Theodor Schnitzler, Philipp Markert, and Markus Dürmuth. "You still use the password after all" – exploring FIDO2 security keys in a small company. In *USENIX 16th Symp. Usable Privacy and Security*, Aug. 2020.

[50] Luca Ferretti, Federico Magnanini, Mauro Andreolini, and Michele Colajanni. Survivable Zero Trust for Cloud Computing Environments. *Computers & Security*, 2021.

[51] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE 30th Computer Security Foundations Symp.*, 2017.

[52] William Fisher, Paul Grassi, Spike Dog, Santos Jha, William Kim, Taylor McCorkill, Joseph Portner, Mark Russell, Sudhi Umarji, and William Barker. Mobile Application Single Sign-On: Improving Authentication for Public Safety First Responders (2nd Draft). Technical report, National Institute of Standards and Technology, 2019. Special Publication 1800-13B.

[53] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword Search and Oblivious Pseudorandom Functions. In *Theory of Cryptography Conference*. Springer, 2005.

[54] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 2014.

[55] Miguel Garcia, Nuno Neves, and Alysson Bessani. SieveQ: A Layered BFT Protection System for Critical Services. *IEEE Trans. Dependable and Secure Computing*, 2016.

[56] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient Middleware for Byzantine Fault Tolerant Database Replication. In *Conf. Computer Systems (EuroSys)*, 2011.

[57] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of Byzantine Agreement in Database Systems. *ACM Trans. Database Systems (TODS)*, 1986.

[58] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers. In *Architecting Dependable Systems II*. Springer, 2004.

[59] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault Tolerance via Diversity for Off-The-Shelf Products: a Study with SQL Database Servers. *IEEE Transactions on Dependable and Secure Computing*, 2007.

[60] Evan Gilman and Doug Barth. *Zero Trust Networks: Building Secure Systems in Untrusted Networks*. " O'Reilly Media, Inc.", 2017.

[61] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *Journal on Computing*, 1988.

[62] Google. Measure performance with the RAIL model, Jun. 2020. `https://web.dev/rail/`.

[63] Paul A Grassi, Michael E Garcia, and James L Fenton. Digital identity guidelines. Technical report, National Institute of Standards and Technology, 2017. DRAFT Special Publication 800-63c.

[64] Shay Gueron and Yehuda Lindell. GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle per Byte. In *Proc. 22nd ACM SIGSAC Conf. Computer and Communications Security*, 2015.

[65] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. In *Annual Int'l Cryptology Conference*. Springer, 1995.

[66] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An Asymmetric PAKE ProtocolSecure Against Pre-Computation Attacks. In *Annual Int'l Conf. Theory and Applications of Cryptographic Techniques*. Springer, 2018.

[67] Ravi Jhawar and Vincenzo Piuri. Fault Tolerance and Resilience in Cloud Computing Environments. In *Computer and information security handbook*. Elsevier, 2017.

[68] Havard Johansen, Dag Johansen, and Robbert van Renesse. Firepatch: Secure and Time-Critical Dissemination of Software Patches. In *Proc. IFIP Int'l Information Security Conf.*, 2007.

[69] Kaspersky Lab. Operation ShadowHammer: new supply chain attack threatens hundreds of thousands of users worldwide. `https://www.kaspersky.com/about/press-releases/2019_operation-shadowhammer-new-supply-chain-attack`, 2019, Accessed Jun. 2020.

[70] John Kindervag et al. Build Security Into Your Network's DNA: The Zero Trust Network Architecture. *Forrester Research Inc*, 2010.

[71] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. Survivable SCADA Via Intrusion-Tolerant Replication. *IEEE Trans. Smart Grid*, 2013.

[72] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre Attacks: Exploiting Speculative Execution. In *Symp. Security and Privacy*. IEEE, 2019.

[73] Diego Kreutz, Alysson Bessani, Eduardo Feitosa, and Hugo Cunha. Towards secure and dependable authentication and authorization infrastructures. In *IEEE 20th Pacific Rim Int'l Symp. Dependable Computing*, 2014.

[74] Trishank Karthik Kuppusamy, Vladimir Diaz, and Justin Cappos. Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories. In *Proc. USENIX Annual Tech. Conf.*, 2017.

[75] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using Delegations to Protect Community Repositories. In *Proc. 13th USENIX Symp. Networked Systems Design and Implementation*, 2016.

[76] Leona Lassak, Annika Hildebrandt, Maximilian Golla, and Blase Ur. "It's Stored, Hopefully, on an Encrypted Server": Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn. In *USENIX Security*, 2021.

[77] Brian Lee, Roman Vanickis, Franklin Rogelio, and Paul Jacob. Situational Awareness based Risk-Adaptable Access Control in Enterprise Networks. In *Proc. Second Int'l Conf. Internet of things, Data and Cloud Computing*. Association for Computing Machinery, 2017.

[78] Allison Lewko and Brent Waters. Decentralizing Attribute-Based Encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011.

[79] Jun Li, Peter L Reiher, and Gerald J Popek. Resilient Self-Organizing Overlay Networks for Security Update Delivery. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.

[80] Chao Lin, Debiao He, Xinyi Huang, Kim-Kwang Raymond Choo, and Athanasios V Vasilakos. BSeIn: A blockchain-based secure mutual authentication with fine-grained access control system for industry 4.0. *Journal of Network and Computer Applications*, 2018.

[81] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[82] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *USENIX Symp. Operating Systems Design and Implementation*, 2016.

[83] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. Byzantine fault-tolerant transaction processing for replicated databases. In *Int'l Symp. Network Computing and Applications*. IEEE, 2011.

[84] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. Mitra: Byzantine Fault-Tolerant Middleware for Transaction Processing on Replicated Databases. *ACM SIGMOD Record*, 2014.

[85] Sanam Ghorbani Lyastani, Michael Schilling, Michaela Neumayr, Michael Backes, and Sven Bugiel. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *IEEE Symp. Security and Privacy*, 2020.

[86] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. Blockchain Based Access Control. In *IFIP Int'l Conf. Distributed Applications and Interoperable Systems*. Springer, 2017.

[87] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. Blockchain based access control services. In *IEEE Int'l Conf. 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.

[88] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. A blockchain based approach for the definition of auditable Access Control systems. *Computers & Security*, 2019.

[89] Federico Magnanini, Luca Ferretti, Mauro Andreolini, Carlo Mazzocca, and Michele Colajanni. SPOC: Survivable Passwordless Single Sign-On. Submitted to *ACM Transactions on Privacy and Security (TOPS)*, 2022.

[90] Federico Magnanini, Luca Ferretti, and Michele Colajanni. Flexible and Survivable Single Sign-On. In Weizhi Meng and Mauro Conti, editors, *Cyberspace Safety and Security*, 2022.

[91] Federico Magnanini, Luca Ferretti, and Michele Colajanni. Scalable, confidential and survivable software updates. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[92] Microsoft Defender ATP Research Team. Windows Defender ATP thwarts Operation WilySupply software supply chain cyberattack. https://www.microsoft.com/security/blog/2017/05/04/windows-defender-atp-thwarts-operation-wilysupply-software-supply-chain-cyberattack/, 2017, Accessed Jun. 2020.

[93] Shigeo Mitsunari. mcl: a portable and fast pairing-based cryptography library. `https://github.com/herumi/mcl`.

[94] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, Nov. 1987.

[95] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *Int'l Conf. Cryptology and Information Security in Latin America*. Springer, 2010.

[96] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *Proc. 26th USENIX Security Symp.*, 2017.

[97] NIST. Software Supply Chain Attacks. `https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter/NCSC_Placemat.pdf`, 2017, Accessed Jun. 2020.

[98] Magnus Nystrom and Burt Kaliski. PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986, 2000.

[99] Rafail Ostrovsky and Moti Yung. How To Withstand Mobile Virus Attacks. In *Proc. Tenth Ann. ACM symp. Principles of distributed computing*, 1991.

[100] Fernando Pedone and Nicolas Schiper. Byzantine fault-tolerant deferred update replication. *Jour. Brazilian Computer Society*, 2012.

[101] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symp. Foundations of Computer Science*. IEEE, 1976.

[102] Indrajit Ray, Kirill Belyaev, Mikhail Strizhov, Dieudonne Mulamba, and Mariappan Rajaram. Secure logging as a service—delegating log management to the cloud. *IEEE Systems Journal*, 2013.

[103] Michael K Reiter, Matthew K Franklin, John B Lacy, and Rebecca N Wright. The Ω Key Management Service. *Journal of Computer Security*, 1996.

[104] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical Report NIST Special Publication (SP) 800-207, National Institute of Standards and Technology, 2020.

[105] Yannis Rouselakis and Brent Waters. Efficient Statically-Secure Large-Universe Multi-Authority Attribute-Based Encryption. In *Proc. Int'l Conf. Financial Cryptography and Data Security*, 2015.

[106] Yumi Sakemi, Tetsutaro Kobayashi, Tsunekazu Saito, and Riad S. Wahby. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-05, Internet Engineering Task Force, Jun. 2020. Work in Progress.

[107] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0, Nov. 2014. `https://openid.net/specs/openid-connect-core-1_0-final.html`.

[108] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable Key Compromise in Software Update Systems. In *Proc. 17th ACM Conf. Computer and Communications Security*, 2010.

[109] Fred B Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990.

[110] Berry Schoenmakers and Meilof Veeningen. Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems. In *Int'l Conf. Applied Cryptography and Network Security*, 2015.

[111] Victor Shoup. Sequences of Games: A Tool for Taming Complexity in Security Proofs. *IACR Cryptol. ePrint Arch.*, 2004:332, 2004.

[112] Karthik Trishank, Brown Akan, Awwad Sebastien, McCoy Damon, Bielawski Russ, Mott Cameron, Lauzon Sam, Weimerskirch André, and Cappos Justin. Uptane: Securing Software Updates for Automobiles. In *The 14th Escar Europe*, 2016.

[113] KD Uttecht. Zero Trust (ZT) Concepts for Federal Government Architectures. Technical report, MASSACHUSETTS INST OF TECH LEXINGTON LEXINGTON United States, 2020.

[114] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling. In *ACM Symp. Operating Systems Principles (SIGOPS)*, 2007.

[115] Romans Vanickis, Paul Jacob, Sohelia Dehghanzadeh, and Brian Lee. Access Control Policy Enforcement for Zero-Trust-Networking. In *29th Irish Signals and Systems Conference (ISSC)*. IEEE, 2018.

[116] W3C. Web Authentication: An API for accessing Public Key Credentials Level 2, 2021. `https://www.w3.org/TR/webauthn-2/`.

[117] W3C. Credential management level 1, 2022. `https://w3c.github.io/webappsec-credential-management/#user-mediated`.

[118] Rory Ward and Betsy Beyer. BeyondCorp: A New Approach to Enterprise Security. *;login:*, 2014.

[119] Thomas Wu, Michael Malkin, and Dan Boneh. Building Intrusion Tolerant Applications. In *USENIX Security Symposium*, 1999.

[120] Yubico. python-fido2, Accessed Jan. 2022. `https://github.com/Yubico/python-fido2`.

[121] Moti Yung. The "Mobile Adversary" Paradigm in Distributed Computation and Systems. In *Proc. ACM Symp. Principles of Distributed Computing*, 2015.

[122] Zirak Zaheer, Hyunseok Chang, Sarit Mukherjee, and Jacobus Van der Merwe. eZTrust: Network-Independent Zero-Trust Perimeterization for Microservices. In *Proc ACM Symp. SDN Research*, 2019.

[123] Yuan Zhang, Chunxiang Xu, Hongwei Li, Kan Yang, Nan Cheng, and Xuemin Sherman Shen. PROTECT: Efficient Password-Pased Threshold Single-Sign-On Authentication for Mobile Users against Perpetual Leakage. *IEEE Transactions on Mobile Computing*, 2020.

[124] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[125] Yan Zhu, Yao Qin, Zhiyuan Zhou, Xiaoxu Song, Guowei Liu, and William Cheng-Chung Chu. Digital Asset Management with Distributed Permission over Blockchain and Attribute-based Access Control. In *IEEE Int'l Conf. Services Computing (SCC)*, 2018.