CryptSDLC: Embedding Cryptographic Engineering into Secure Software Development Lifecycle

ABSTRACT

Application development for the cloud is already challenging because of the complexity caused by the ubiquitous, interconnected, and scalable nature of the cloud paradigm. But when modern secure and privacy aware cloud applications require the integration of cryptographic algorithms, developers even need to face additional challenges: An incorrect application may not only lead to a loss of the intended strong security properties but may also open up additional loopholes for potential breaches some time in the near or far future. To avoid these pitfalls and to achieve dependable security and privacy by design, cryptography needs to be systematically designed into the software, and from scratch. We present a system architecture providing a practical abstraction for the many specialists involved in such a development process, plus a suitable cryptographic software development life cycle methodology on top of the architecture. The methodology is complemented with additional tools supporting structured inter–domain communication and thus the generation of consistent results: cloud security and privacy patterns, and modelling of cloud service level agreements. We conclude with an assessment of the use of the Cryptographic Software Design Life Cycle (*CryptSDLC*) in a EU research project.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software development methods**;

KEYWORDS

Software engineering, Software design life cycle, Cryptography, Security by design, Privacy by design, Data protection by design and default

ACM Reference Format:
. 2018. CryptSDLC: Embedding Cryptographic Engineering into Secure Software Development Lifecycle. In *Proceedings of ARES (ARES'18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Developing secure software is one of the most challenging tasks in software engineering and it is ever becoming more important. Novel technologies like cloud computing and Internet of Things are leading to highly distributed interconnected systems communicating over the Internet—thus adding further problems to the so–called trinity of trouble: *connectedness, complexity, and extensibility*.

Security by design is the new paradigm for developing secure software. It can basically be considered as a set of best practices embedded into a software development lifecycle (SDLC). It is a secure software development lifecycle (SSDLC) which integrates security considerations into all phases of the development process.

Data protection by design and default is another related concept[1] which needs to be put into practice. In a very informal way we could say that data protection by design covers the security of personal data[2]. Obviously many concepts from secure application design–e.g. as regards confidentiality of data, or hardening of information and communication technology systems–may be applied in privacy aware systems. Other cryptographic technologies, e.g. anonymous authentication, or redaction of digitally signed documents for data minimization purposes can be applied in a similar framework as other security technologies that do not fulfill privacy functions for the protection of personal data in the first place.

Yet both security and privacy need to be integrated from the very beginning into a software development process [23]. Many of the problems often encountered in the development phase can be mitigated on the architectural level if security aspects are already considered in the design phase. This is specifically true when cryptography is used.

Cryptography can help on many levels to efficiently and effectively protect data and information systems. It is a preventive technology that needs to be carefully designed into the system from scratch to unleash its full potential. When cryptography is not correctly applied it may not only lose its protective properties but even introduce new vulnerabilities. Despite the importance of this topic we did not find adequate treatment in literature. In this paper we will explain why cryptography needs to be considered systematically during the entire lifecycle of the software development process—and specifically during the design phase—and present a first conceptual approach how this can be practically achieved. To the best of our knowledge, this is the first systematic and holistic approach aiming at an integration of *cryptographic engineering* with a *secure software development process*.

In section 2 we review the state-of-the-art and identify challenges we encountered during our work in development projects involving a broad number of specialists from different disciplines. In section 3 we introduce a system architecture providing a tangible abstraction for the multiple disciplines involved in a cryptographic

[1]'Data protection by design and by default', as it is defined in the EU GDPR [11], Art. 25 (cf. http://www.privacy-regulation.eu/en/article-25-data-protection-by-design-and-by-default-GDPR.htm (accessed May 2018) is the more present term for what is still colloquially referred to as 'information privacy', or in US legal environment 'protection of personally identifiable information (PII)'

[2]Art. 4, Par. (1), EU GDPR[11] defines personal data meaning "any information relating to an identified or identifiable natural person ('data subject')"

development process. The architecture is accompanied by the novel methodology for secure service design and development *CryptSDLC* (section 4). Finally, methods for inter domain communication based on *cloud security and privacy patterns* and *capabilities and SLA modeling* are presented in section 5. Finally (section 6) we give a critical assessment of our practical experience with the *CryptSDLC*.

2 CHALLENGES

The necessity of integrated secure software development has been established and several approaches were proposed. Nevertheless, methodologies still do not sufficiently cover the practical adaptation and integration of cryptographic primitives and protocols in software development life cycles. In the following we present a state-of-the-art and identify additional supporting principles that we intend to address with our newly proposed *CryptSDLC*.

2.1 State of the art and related work

The importance of preventing software security defects is now widely accepted. The first books about these topics appeared around 2001 [24]. Software security best practices involve thinking about security early in the development life cycle and embed security consideration in all phases of the life cycle. Surveys of whole life cycle practices and life cycle phase-specific practices can be found in [8], [18], [24] and [30].

Also industry is catching up fast to cope with the new situation. Microsoft has carried out a noteworthy effort and developed The Security Development Lifecycle (SDL) [15]. SDL is one of the industry standards today, because it is open and free tool support exists for some steps in the process.

Furthermore, the Open Web Application Security Project (OWASP) community developed the Software Assurance Maturity Model (SAMM) [10] which is a more lightweight analog to the Microsoft approach. SAMM helps organizations assess, formulate, and implement a strategy for software security, that can be integrated into their existing Software Development Lifecycle (SDLC).

Besides industry and community approaches taking momentum, standardization bodies also started efforts to tackle the problem in a systematic way. ISO/IEC 27034 offers guidance on information security to those specifying, designing and programming or procuring, implementing and using application systems, in other words business and IT managers, developers and auditors, and ultimately the end-users of ICT. The aim is to ensure that computer applications deliver the desired or necessary level of security in support of the organization's Information Security Management System, adequately addressing many ICT security risks.[3] Microsoft also has declared conformance with ISO 27034-1, the first part of a relatively new international standard for secure software development.

2.2 Towards Cryptographic Engineering

In summary, comprehensive approaches to secure software development exist but they do not explicitly cover the integration of cryptographic engineering into the secure SDLC. Only little is known about the incorporation of cryptographic engineering into the processes developed, even though cryptographic solutions are acknowledged as being important and versatile technical solutions to protect information assets. For example, in SDL the recommendations for use of cryptography[4] are very basic and do not cover any advanced primitives or protocols. By cryptographic engineering we mean the selection, adaption, implementation, and roll-out of cryptographic primitives in productive systems. This goes beyond enabling known existing methods, like TLS on a communication link between two back-end servers.

In support of this rationale, the very recent results in [25] clearly indicate the difficulties in developing cryptographic applications (by analyzing software development approaches to password storage). The most important outcome was, that developers think of functionality first, before they think of security. Developers need to be reminded of this topic before the start of a project. Even worse, none of the produced solutions met current academic standards–mainly because the they were not integrated with the frameworks widely used in development.The same is true for other primitives and protocols, e.g, as shown in [29].

Even if the engineering process does not involve a (re)design of cryptographic algorithms, as in the case of using SSL or TLS–which are based on existing cryptographic primitives and standardised protocol descriptions–Google found the need to re-implement the protocols as they were not content with the current implementations[5]. A first approach towards the integration of cryptographic engineering has been presented in [2]. There are now also first commercial products available, e.g. Cryptosense[6]. However, the approach is very limited and does not cover the design phase of the lifecycle, but only targets at the development phase.

2.3 Supporting principles

During development and practical application of the *CryptSDLC* we identified several inhibitors and promoters of a successful integration of cryptographic protection into cloud services [22]. From the promoters we derived three principles that need to be supported in the cryptographic software design life cycle.

2.3.1 Cryptographic engineering needs to be closely coupled with today's flexible and agile development. Historically, cryptography works in a reliable way for well confined, understood problems and in static scenarios. Most prominently it was used to establish secure channels over untrusted networks and to protect data at rest. All these tasks have well defined security requirements which are not supposed to change over time or per application. The same is true for the underlying trust models. However, from the past we learned that even such standard designs can miserably fail when done in an ad-hoc fashion. An example is the wrong combination of encryption (ENC) modes and message authentication codes (MAC) leading to padding oracle attacks [19] which allowed breaking the otherwise secure encryption scheme inside SSL. Thus cryptographic engineering needs to be closely coupled with today's flexible and agile development.

[3]http://www.iso27001security.com/html/27034.html, accessed May 2018.

[4]https://download.microsoft.com/download/6/3/A/63AFA3DF-BB84-4B38-8704-B27605B99DA7/Microsoft%20SDL%20Cryptographic%20Recommendations.pdf, accessed May 2018.

[5]For Google's BoringSSL effort see https://www.imperialviolet.org/2014/06/20/boringssl.html and for Go a dedicated implementation exists as well see https://groups.google.com/forum/#!topic/golang-nuts/0za-R3wVaeQ, accessed May 2018.

[6]https://cryptosense.com/analyzer/, accessed April 2018

2.3.2 Cryptographic engineering starts from a deep understanding of the cryptographic primitives. Since the turn of the millennium, cryptographic research has developed a large body of primitives and protocols providing the potential to add functionality and flexibility for cloud applications over the entire data lifecycle (cf. e.g. [22] . But engineering these advanced cryptographic functionalities into software is a very complex task and requires deep cryptographic expertise. And while contemporary secure software development life cycles recommend the use of cryptography, only little actual guidance exists on how secure adaption and secure integration can be achieved in practice.

2.3.3 Cryptographic engineering requires interdisciplinary expertise and communication. Engineering advanced cryptographic functionality into software systems requires expertise from different fields: On the lowest level cryptographic expertise and the expertise to actually securely implement the cryptography is needed. Often these skill are not found in the same set of people, and interdisciplinary collaborations among cryptographers and crypto developers are required. Protection requirements are usually coming from service and application designers and need to be communicated to cryptographers and crypto developers. On the other hand, capabilities and potentials of cryptographic algoriths need to be communicated to the higher level service and application designers. Security experts overseeing the secure integration need to be active on all levels of a development process.

3 ARCHITECTURE

The main goal of the *CryptSDLC* architecture is to support the development process of cryptographic applications. It is a conceptual approach to structure and streamline typical tasks identified in cryptographic engineering. The model helps to cope with the complexity and interdisciplinary nature of cryptographic application design. It is based on the experiences made in an EU research project with people from very different disciplines involved, all targeted at a single goal: the development of cloud services which are secure by design and leverage feasible cryptography.

The development of the architecture was driven by two factors: On one hand, we needed a tool to improve communication between the experts of the different disciplines involved in our projects. On the other hand, it was a basis for the development of the accompanying methodology which relies on the communication structure defined in the architecture with it's layers and interfaces. Especially the interface between cryptographic researchers and software architects turned out to be essential to ensure that security properties developed at the cryptographic layer can be transferred to cloud applications.

In particular, the goals of the architecture are: incorporation of cryptographically sound design methodologies; support adoption by efficient and secure implementations of generic building blocks; give guidance for use of cryptography in a developer friendly way; reduce configuration and integration errors as far as possible without limiting the flexibility; foster exploitation of results for all application domains (horizontal technology); and enable fast adoption for a large developer base.

3.1 Architecture Layers

The *CryptSDLC* architecture is shown in Figure 1 and comprises four different layers, three of which are of major interest for this document, namely: primitives, tools and services. Nevertheless, we subsequently introduce all four and quickly present the main ideas behind.

Primitives Layers. The lowest layer consists of *cryptographic primitives and protocols* which represent basic cryptographic building blocks, e.g., signature schemes, or cryptographic protocols. They typically provide very specific functionality which is defined by the security goals achieved. Furthermore, the analysis and development of cryptographic primitives is done by cryptographers with strong mathematical background, ideally in a provable manner, i.e., by rigorous mathematical methods and models. The very specialized knowledge required for this work is not shared by software developers.

In Figure 1 some of the cryptographic primitives covered in the research project are shown which serve as an example in our work. Note, research on the cryptographic layer is essential for building cryptographically enhanced services in order to provide the required functionality and efficiency for application usage. The work conducted on the primitives layer typically aims at closing the gaps derived in the requirements engineering process from use cases and their services.

Tools Layer. The second layer is denoted as *tools* layer. Tools are a concept introduced by *CryptSDLC* to communicate techniques to software developers and architects in an more accessible way. They provide higher level functionality as a combination of primitives which serve a particular purpose and also come with an accompanying implementation in form of, e.g., software libraries. The design of tools is still based on rigorous cryptographic models and ideally provides provable security for realistic adversary models. For software developers the tools layer must provide all documentation to use and integrate the tool libraries correctly into services. In should translate all essential information from the mathematical world to a more accessible form for practitioners.

Service Layer. Cloud computing is radically changing the way we are consuming IT resources but also influencing the way software is built and deployed. Although not really new, services and microservices based design is becoming a dominant pattern in industry [26] to increase flexibility and reuse of components. Away from monolithic architectures, we are facing a shift towards service oriented architectures (SOA) where services can be flexibly interconnected and deployed in distributed fashion spreading traditional perimeters. In *CryptSDLC* we acknowledge this trend and its benefits by the definition of the service layer which is extremely useful in the context of cryptographic engineering to further encapsulate more complex tasks from application developers.

A *CryptSDLC* service can be seen as a customization of a particular tool for one specific application—thus we call a service an instantiation of a tool. Thus, a service is a way to deliver the tool to system and application developers, the users of the tools, in an preconfigured and accessible form. They will be able to integrate the services without deeper understanding of tools and primitives and ideally without even being an IT security expert. However,

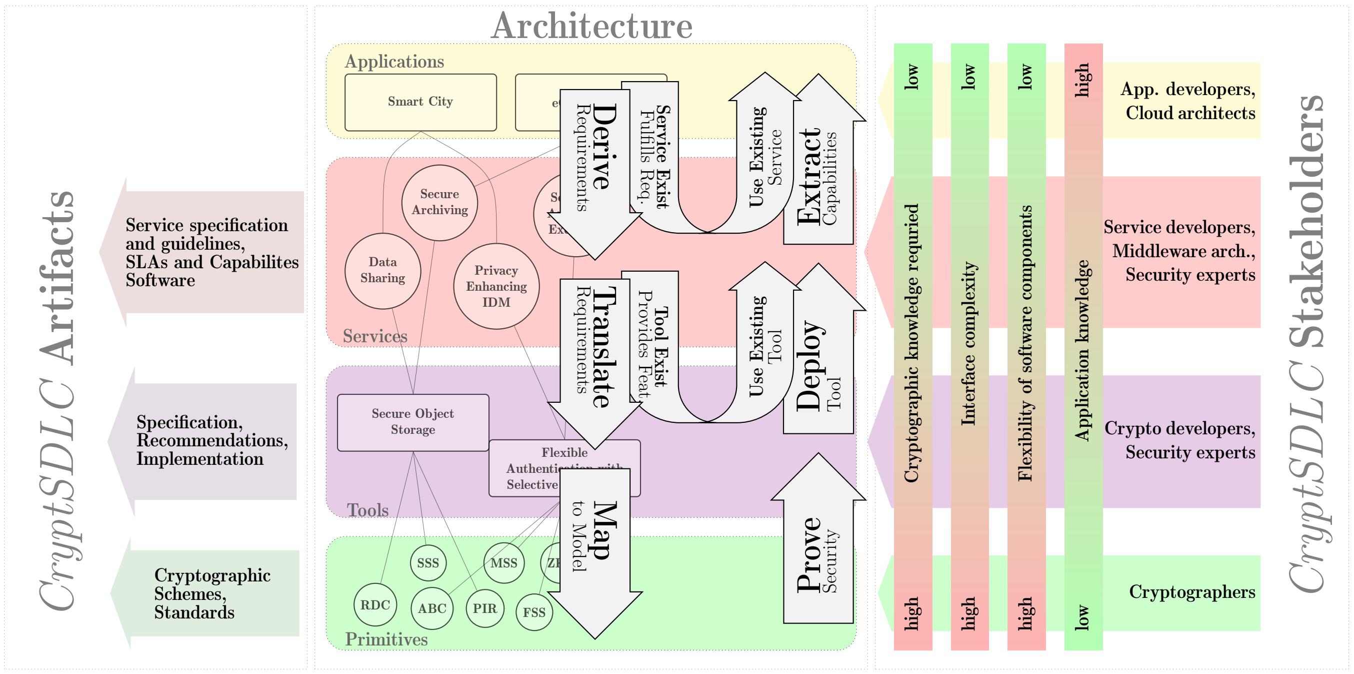
CryptSDLC Artifacts
Service specification and guidelines, SLAs and Capabilites Software
Specification, Recommendations, Implementation
Cryptographic Schemes, Standards
Architecture
Applications
Smart City
Services
Secure Archiving
Data Sharing
Privacy Enhancing IDM
Tools
Secure Object Storage
Flexible Authentication with Selective
Primitives
SSS
MSS
RDC
ABC
PIR
FSS
Derive Requirements
Service Exist Fulfills Req.
Use Existing Service
Extract Capabilities
Translate Requirements
Tool Exist Provides Feat.
Use Existing Tool
Deploy Tool
Map to Model
Prove Security
Cryptographic knowledge requried
Interface complexity
Flexibility of software components
Application knowledge
low
high
App. developers, Cloud architects
Service developers, Middleware arch., Security experts
Crypto developers, Security experts
Cryptographers
CryptSDLC Stakeholders

Figure 1: The *CryptSDLC* Architecture and Methodology overview.

because the services are built atop sound cryptographic concepts, i.e., out of the *CryptSDLC* tools, they provide strong security guarantees and are built the right way. This approach is different to the broadly applied ad-hoc integration of cryptographic solutions into applications and helps to avoid common pitfalls in system design and implementation.

Application Layer. The application layer contains the applications targeted at real end users. Modern applications try to leverage the idea of service oriented architectures to support scalability and elasticity through modularization. This gives the required freedom in deployment needed in modern cloud environment, be it private, public or hybrid cloud settings. The concept of *CryptSDLC* follows this trend and the services based approach are a modern way to expose security functionality to cloud architects and application developers.

3.2 Closing the Gaps

The structure of the architecture was designed in order to mitigate all three gaps identified earlier: The architecture incorporates a set of suitable cryptographic primitives first into tools, then into services and finally into an application. It describes how to add security functionality at the lowest possible layer. This positively impacts the security of the application and thus bridges the 1^{st} gap by coupling the software development process with cryptographic engineering as well as highlighting the need for a deep cryptographic understanding (2^{nd} gap). We believe that if the security functionality is not addressed at the lower layer, but addressed by alternative security controls on higher layers, it introduces additional overhead in complexity and administration, which can also lower the security of the system.

Moreover, the architecture separates certain areas of expertise or describes the need for cooperation and communication (3^{rd} gap) among different disciplines: For example cryptographers and software developers work together to code *CryptSDLC*'s tools. This is clearly highlighted by the architecture. Those tools encapsulate the inner workings of the cryptographic functionality and offer a defined software interface to a set of specifically tailored algorithms serving a dedicated purpose. Having the tools layer, a service can be built in less time by less specialized software developers, especially it needs only a limited level of knowledge in cryptography. Thus, building cryptographic services out of those tools becomes much easier as it requires much less cryptographic knowledge than without this intermediate step. With tools being the software or hardware that computes the cryptographic algorithm or protocol, the tool is much more flexible and its many more options can be customized to provide several services. Then each service has many potential applications in which it can be re-used to targeted customers' and business' domain-specific security or data-protection requirements. This again addresses the 1^{st} gap as it allows for an agile and flexible re-use of cryptography-based security functionality and allows formal modelling to the highest level possible with reasonable effort.

4 CRYPTSDLC: A NEW DEVELOPMENT METHODOLOGY

In the following we present a first version of our methodology specifically covering the cryptographic engineering aspects in SSDLCs. It is called *Cryptographic Software Development Lifecycle (CryptSDLC)* and defines a way to traverse the layers in the architecture. It can be seen as an extension to SSDLC and shall assist software developers to get cryptographic usage right in their applications. As a major contribution it standardizes the steps necessary when going from one layer to the other and aligns them to the general phases of classical SSDLC models.

The major steps of CryptSDLC have already been included in Figure 1 and are marked by the big white arrows in the middle. The major steps are *Derive, Translate and Map* from top to bottom and *Prove, Deploy and Extract* from bottom to top. The *CryptSDLC* can be used to extend conventional software development lifecycles, like the ones presented in section 2. It basically introduces an additional dimension coping with cryptographic engineering and has a strong focus in the requirements and design phase. However, also the development and deployment phases are of importance but omitted in this work for space reasons.

In the requirements phase the following steps give a top-down approach for cryptographic requirements gathering:

- **Derive Requirements:** Based on the requirements gathered we derive the most important ones for the core cryptographic service we want to use or build.
- **Translate Requirements:** The requirements are translated into a more formal language which can be understood and used by cryptographers to start their research and design.
- **Map to Model:** To trigger research on primitives and protocols the identified gaps on the tools layer have to be mapped to research goals in cryptography for specific primitives or protocols.

In the design and development phases we use a bottom-up approach and define the following steps to go up in the *CryptSDLC* architecture:

- **Prove Security:** Tool should be built by formal methods used in cryptography as good as possible. The goal is to support features by the definition of provably secure protocols.
- **Deploy Tool:** The components of a tool are arranged in the service architecture in a way such that by reasoning it gets clear how the features of the tool translate the security requirements fulfilled by the service.
- **Extract Capabilities:** Based on the features of the tool and the deployment model specific service properties called capabilities can be extracted. They are exposed as an additional property to the upper application layer.

Although we define this holistic approach going down to the lowest layer, the reuse of existing work is a major goal of the whole process. In all layers the step down to lower layers is only performed if the requirements cannot be already achieved with available solutions by the following policy:

- *Application layer:* Only develop a dedicated service if requirements cannot be fulfilled otherwise.
- *Service layer:* If possible, develop the new service on the basis of existing templates and just add missing features supported by an underlying tool.
- *Tool layer:* Only develop a new tool if given tools cannot provide the required features or an existing tool cannot be extended with the required feature.
- *Primitive layer:* Always do a state of the art analysis if missing features can be provided by existing research results or specify the gap if not. Only then trigger research activities on the primitive layer.

The top down path is basically a very detailed requirements analysis process decomposing and breaking down the various requirements down to a level where cryptographers can work with them. The bottom up path then focus on the composition of upper layer components based on the functionalities available in the lower layers. The full cycle does not always to be walked through. In fact, one should only go down one layer when exiting solutions do not fulfill the requirements at the given level and need some modifications to provide the required features. *CryptSDLC* has been implemented and successfully tested within a research project.

However, in the remaining we will focus on the analysis of the very new steps "Proof" and "Deploy" in this section, which are new steps to be considered in secure development cycles during the design phase. We will also not look into testing and deployment phases, which is part of future work.

4.1 Proof: Compose primitives into tools

Due to the cryptographic nature of the tools it is of prime importance to have a profound analysis and sound security proofs for them. In the following we shortly discuss the two main approaches being used in cryptography for composing primitives to tools.

Universal Composability is a very rigorous approach used to overcome the problem of protocols being analyzed as standalone applications. It enables the construction of security models where security is retained under protocol composition which is the most generic result to achieve. Multiple instances of the protocols can run concurrently or even interact with each other without violating the security model. Various frameworks can be found in the literature, e.g., Canetti, Hirt and Maurer, Pfitzmann et al., and Küsters et al. [5–7, 20]. In *CryptSDLC* the advantages of constructing tools in universal composability frameworks is that they can be combined and the security of the combination follows. However, it comes with a number of drawbacks, e.g. sometimes impractically high computational overheads. Thus, designing UC-secure tools requires effort on the primitives level in order to get efficient building blocks that can be composed in a modular way. Many issues are currently getting addressed in academic research, see for example [4], so we recommend it for future use or simpler protocols.

Direct Construction of High-Level Primitives is the other main approach for constructing complex primitives and tools and proving them secure are direct (or ad hoc) constructions. That is, one defines a set of experiments covering the security properties one wants to realize with the given functionality, e.g., unforgeability of signatures, confidentiality against a defined class of adversaries for encryption, etc. One then specifies concrete instantiations of algorithms and proves that those algorithms indeed satisfy those security definitions. The main advantage of ad hoc constructions is an increased efficiency compared to universally composable constructions, and sometimes the only way to proof security when UC approaches fail. However, one of the most fundamental drawbacks is that security is typically not retained under concurrent composition and security proofs tend to be monolithic and non-modular.

A general aspect of the *Proof* phase is, that although techniques from provable security are used, there is always an abstraction step involved on how the environment is modeled. Thus, the security models used can not cover all aspects and existing gaps and their implications have to be communicated to the software developer. It is important that it's clear to the users of the tool under which conditions the security properties hold and how they map to real world scenarios. Examples for such assumptions are parties which are considered honest-but-curious or synchronous network models which need additional security controls on higher levels to be assured in specific use cases.

Finally, the implementation of the tool has to be considered during this phase. Providing secure and trustworthy implementations of core components which enable the very features defined in the tool specification is essential for the next step. The development of reliable core components in software and hardware is a very challenging task and requires a lot of experience. The developers must be able to understand the theoretical side, but also be aware of implementation challenges from real world settings, e.g., how to implement side channel resistant code. Although current approaches mainly follow a heuristic approach, for the future we envisage the ability to proof the correctness of code with methods from formal verification to establish a so called trusted code base (TCB) as in [14].

4.2 Deploy: Compose tools into services

The tool abstraction we have introduced in *CryptSDLC* is a key concept which greatly simplifies the development of secure services and should also provide better results in terms of secure design. It is also intended to maximizes the reuse of existing work on the cryptographic layer. The process of generating a service out of a tool has to cover all additional steps not covered by the tools but needed for real world applications. In particular, the following steps are necessary to design a service out of a tool:

(1) Specify a service & deployment plan incl. stakeholders
(2) Identify major components in the service and sketch their main functionality
(3) Embed tool components within service components
(4) Map requirements to features provided by the tool
(5) Generate a software architecture and specification
(6) Implement software development lifecycles with integrated security
(7) Propose operational guidelines like an assurance model to support production phase

Using the proposed methodology should lead to services which are secure by design and built on cryptographically sound composition of primitives and protocols, i.e., in a provable way in the best case. Especially, after embedding the components of the tool into the components of the service—the deployment step—and considering all additional guidelines specified by the tool, e.g. "communication between server and dealer must be private and authentic", we can reason about the service to be a secure instantiation of the tool.

Naturally, when building a piece of software there are additional aspects to consider apart from using correct algorithms, i.e., correct cryptography in our case. It faces all challenges known from secure software development and all state-of-the-art processes and methodologies for SDL shall also be applied during the *CryptSDLC* service development. Tools also offer an additional benefit: increased development speed and improved security through the secure and efficient software software implementations of core cryptographic functions provided with the tools.

Nevertheless, a complete service is comprised of many dedicated software components running within the cloud infrastructures. Even worse, operational aspects have to be considered and defined to fully support a cloud service life cycle thus mandating integration into operational processes. In summary, although the tool concept greatly facilitates the service development process, all SDL documentation must be heeded to foster quick adoption of project results on the service level.

4.3 Extract: Advertise a service's impact

A Service Capability in *CryptSDLC* is a security & privacy relevant property that is of importance for the application domain of the service and that can be described in a specific and precise language, and additionally in a formal, machine readable language. Service capabilities can and should thus be used to advertise towards the application level what the service is able to provide. They shall facilitate comparisons among services with respect to specific properties as well as guide the communication of experts from different fields (see Sec. 5.1). Thus, they should highlight positive impacts, e.g. 'service achieves confidentially of stored data', as well as additional overheads, e.g. 'service requires twice as much computations as a regular asymmetric integrity protection'. They should be best contractually agreed upon, i.e. be part of a service level agreement (SLA), see Sec. 5.3.

5 INTER STAKEHOLDER COMMUNICATION

As shown in Fig. 1, the four tiers of the architecture span across domains with different stakeholders involved (i.e. the cryptographers, crypto developers, service developers, application developers etc). We provide three communication tools to close communication gaps between experts involved on the different layers:

- *Capabilities:* A *service capability* is a security & privacy relevant property that is of importance for the application domain of the service and that can be described in a precise language—at best a machine readable language in order to enable comparisons among services with respect to specific properties. With an English language description this forms a dictionary for communication among the experts for properties like 'availability' or 'confidentiality of data towards the storage provider'. The challenge is to have them remain precise enough to still capture only the cryptographically proven security claim while still abstractly communicate enhancements towards the service consumers.
- *Patterns:* In order to better understand which services to incorporate into an application we propose *cloud security and privacy patterns* and *human computer interaction (HCI) patterns.* Each pattern describes the problem a certain service can solve and how and with which implications that can be achieved.
- *Service Level Agreements:* Finally, we advertise the positive impacts of involving cryptographic primitives in applications, but also their overheads, on the level of services as *service level agreements (SLAs).* We base them around the capabilities—and standardised property descriptions—allowing application users, e.g.

cloud service customers, to understand them without requiring the full knowledge of the cryptographer—who designed and established the security of a cryptographic primitive.

These communication tools bridge existing gaps and support the creation of sound cryptographically-enhanced cloud services concisely catering for the end users' security and data protection demands.

5.1 Service Capabilities

We identified early that the involved stakeholders do not always speak a common language. Let us recall one extreme example that occurred in the research project: In cryptography—in the domain of information privacy—there were publications that achieved a 'hiding property' the researchers called 'transparency', thus redefining a name used in the legal domain with an entirely different, if not opposite, meaning (i.e. allowing successful inspection of all details)[7]. With a precise definition of a property that is either required or achieved, the *CryptSDLC* workflow can be initiated either top-down or bottom-up. For example in the bottom-up case, a Service Capability can be used to advertise a service's cryptographically-proven data-protection increase as an added value to the market more easily. In order to keep the linkage between the actual achievement and the actual tools and thus the cryptographic primitive's security gains, the service capability is not just stated statically, but a model is provided that derives such a capability from the configuration parameters of their cryptographic software libraries – called tool in the architecture. This model encapsulates then the transformation along the bottom-up steps described *CryptSDLC*. Furthermore, one can model the transformation process in a top-down direction: As a result one is able to adapt the configuration and inner workings of the cryptographic underpinnings of a service swiftly to address changing customer demands. Together with a model the service capability holistically captures the relation between cryptographic parameters—lowest-level in the *CryptSDLC* architecture—and high-level customer requirements.

5.2 Design Patterns

In order to support the communication within the layered development process governed by *CryptSDLC*, as well as to support the diffusion of novel cryptographic paradigms and capabilities among prospective providers and end users we propose *cloud security and privacy patterns* and *HCI–human computer interaction–patterns*. Both cloud security and privacy patterns and HCI patterns are used to codify expert knowledge and requirements within a specific scope in a way that the information remains accessible across domains of involved actors. The main idea is that a design pattern shall "describe(s) a problem which occurs over and over again (...) and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over (. . .)" [1]. This is done by describing the (empirical) background of the pattern, i.e. the "problem", and giving instructions for the "solution" in natural language in a framework of categories.

The concept was invented in Berkeley, CA., in the 1970s for application in architectural design [1] and has later on been modified for application in software architecture in the 1990s when object oriented design and re-usability required efficient communication of complex issues across different domains of involved people [13]. Later on, the concept was used for security and privacy patterns [9, 28], as well as for human computer interaction aspects in HCI patterns [12][21]. Since several years, there exist collections and catalogues of cloud security and privacy patterns specifically for modelling threats and solutions in the cloud context.

In the research project we have proposed a set of nine cloud security and privacy patterns[8]covering approximately the cases for which we intended to provide secure implementations by project end. We used these original patterns as common reference and communication means between tools developers, service developers, and application developers in the actual process of developing the architecture and the *CryptSDLC* method.

5.3 SLA Modelling

With service capabilities describing already properties of interest it was interesting to map those security and data-protection-enhancing cryptographically supported properties into the formal agreements that govern the relation between the upper two layers of the *CryptSDLC* architecture: Service Level Agreements (SLAs) between applications and services. Like service capabilities SLAs are the interface by which services are advertised and they shall facilitate the comparison between similar offerings. Not surprisingly for the cloud services an internationally agreed set of terms is becoming an ISO standard, i.e. ISO/IEC 19086 [16]. There is also a subpart of the standardised SLA component that deal with security and privacy[9]. To give a concrete example, there is the suggestion to have an SLA component that describes the 'Cryptographic controls for data at rest' in ISO/IEC 19086-4. The standardised component covers "[...] the cryptographic controls available for data at rest associated with the covered services. Note: These controls provide for securing data with respect to confidentiality and integrity, while being stored in a covered service. [...]" [17]. As shown in this case the SLA's contents can be mapped onto one of the previously mentioned service capability and by that the correct cryptographically enhanced service can be chosen. The idea is that the SLA will further denote which cryptographic algorithms and security parameters are used, such that instead of void marketing clauses of "banking grade encryption" it would be common in the market to name the algorithm, e.g. '3DES'[10]. Thus, SLA's facilitate the communication of the services towards the higher layers of applications and finally also the end-consumers, e.g. humans using the applications.

6 CONCLUSIONS

In this work we summarized the problems encountered in cryptographic engineering and proposed a first solution towards a systematic integration into a secure SDLC. The proposed *CryptSDLC* methodology addresses communication issues between different stakeholders needed to interweave the development of cryptographic solutions as foundations for secure applications, offering data-protection by design and default. The goal of the current work

[7]See for example the cryptographic adjustments made by Brzuska et al. [3] following the legal discussion in Pöhls et al. [27]

[8]Will be cited in de-anonymized paper todo

[9]Privacy in the sense of protection of personal data if translated from international to EU GDPR 'language'

[10]This is indeed in many banking systems, e.g. banking cards, a still used algorithm

is to give software developers access to advanced cryptographic solutions and to reduce potential error sources for software developers integrating cryptographic functionality in their design.

The presented approach was developed and successfully tested in a large EU research project which had all relevant stakeholders on board. Although the feasibility of the approach was demonstrated in the field of cloud computing, it can be considered generic enough to become suitable for modern application design also in other paradigms.

This is the first step towards a full-fledged integration of cryptographic engineering in a secure software design life cycle. In the current version of *CryptSDLC* documentation and software artifacts are mainly defined for the requirements and development phases. A first step was done also to cope with the deployment phase, however, more work needs be done into this direction as well as towards a tighter automated toolchain integration.

Additional work is required to further improve the handling of cryptographic topics. Standardisation of cryptographic primitives and their properties needs to be advanced (to have a precise language as a common denominator). One particular problem is that access to cryptographic expertise is only available in large companies or specialized departments of universities. Typically software developers only have limited access to this cryptographic know-how within their environment. However, if tool level modifications are necessary it is inevitable to involve this expertise, and if not internally available, it should be contracted for limited time.

To avoid several of these problems, we envisage a community approach based on open technologies and information sharing. Ideally, relevant information about tools and services is collected in a public repository providing software developers easy access to state-of-the-art tools and services, as well as to academic knowledge. This can be the contact point where the cryptography community and the security and software development experts can exchange their knowledge. The platform could work on similar principles as *bettercrypto.org*, which helps IT administrators without in-depth knowledge to securely configure their systems.

REFERENCES

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

[2] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. 2015. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) - Onward! 2015.* ACM Press, New York, New York, USA, 1–13. https://doi.org/10.1145/2814228.2814229

[3] Christina Brzuska, Henrich C. Pöhls, and Kai Samelin. 2012. Non-Interactive Public Accountability for Sanitizable Signatures. In *Revised Selected Papers of European PKI Workshop: Research and Applications (EuroPKI 2012) (LNCS)*, Vol. 7868. Springer, 178–193. http://dx.doi.org/10.1007/978-3-642-40012-4_12

[4] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2016. Universal Composition with Responsive Environments. *IACR Cryptology ePrint Archive* 2016 (2016), 34. http://eprint.iacr.org/2016/034

[5] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001.* IEEE Computer Society, 136–145.

[6] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. 2006. Time-Bounded Task-PIOAs: A Framework for Analyzing Security Protocols. In *Distributed Computing, 20th International Symposium, DISC 2006 (Lecture Notes in Computer Science)*, Shlomi Dolev (Ed.), Vol. 4167. Springer, 238–253.

[7] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007 (Lecture Notes in Computer Science)*, Salil P. Vadhan (Ed.), Vol. 4392. Springer, 61–85.

[8] Noopur Davis, Watts Humphrey, Samuel T. Redwine, Gerlinde Zibulski, and Gary McGraw. 2004. Processes for producing secure software: Summary of US national Cybersecurity Summit subgroup report. *IEEE Security and Privacy* 2, 3 (may 2004), 18–25. https://doi.org/10.1109/MSP.2004.21

[9] Nick Doty and Mohit Gupta. 2013. Privacy Design Patterns and Anti-Patterns. In *Workshop "A Turn for the Worse: Trustbusters for User Interfaces Workshop" at SOUPS 2013 Newcastle, UK.*

[10] Pravir. et al. Chandra. 2009. Software Assurance Maturity Model. (2009), 96 pages. https://www.owasp.org/index.php/OWASP{_}SAMM{_}Project

[11] European Commission. 2016. Regulation (EU) 2016/679 of The European Parliament and of The Council, of 27 April 2016, on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). (2016). http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679 (online 20.7.2017).

[12] S. Fischer-Hübnner, C. Köffel, J.-S. Pettersson, P. Wolkerstorfer, C. Graf, and L. Holtz. 2011. HCI Pattern Collection–Version 2. (2011). http://primelife.ercim.eu/results/documents/111-413d (PrimeLife project).

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. ISBN 0-201-63361-2.

[14] Harry Halpin. 2018. A Roadmap for High Assurance Cryptography. Springer, Cham, 83–91. https://doi.org/10.1007/978-3-319-75650-9_6

[15] Michael Howard and Steve. Lipner. 2006. *The security development lifecycle : SDL, a process for developing demonstrably more secure software.* Microsoft Press.

[16] ISO. 2016. *Information technology — Cloud computing — Service level agreement (SLA) framework and technology — Part 1: Overview and concepts.* Standard. International Organization for Standardization, International Electrotechnical Commission. Final Draft.

[17] ISO. 2017. *Information technology — Cloud computing — Service level agreement (SLA) framework — Part 4: Components of Security and of Protection of PII.* Standard. International Organization for Standardization, International Electrotechnical Commission. Committee Draft.

[18] K R Jayaram and Aditya P Mathur. 2005. *Software Engineering for Secure Software - State of the Art: A Survey.* Technical Report. Purdue University. 1–35 pages. papers3://publication/uuid/f46d5eb9-0f9d-4b70-a8da-34c0c6b046ab

[19] Hugo Krawczyk. 2001. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Annual International Cryptology Conference.* Springer, 310–331.

[20] Ralf Küsters and Max Tuengerthal. 2013. The IITM Model: a Simple and Expressive Model for Universal Composability. *IACR Cryptology ePrint Archive* 2013 (2013), 25.

[21] Thomas Länger, Ala Alaqra, Simone Fischer Hübner, Erik Framner, John-Sören Pettersson, and Katrin Riemer. 2018. HCI Patterns for Cryptographically Equipped Cloud Services. *Springer LNCS, Proceedings of the HCI International 2018 âĂŞ 20th International Conference on Human-Computer Interaction, Las Vegas, USA* (2018).

[22] Thomas Lorünser, Stephan Krenn, Christoph Striecks, and Thomas Länger. 2017. Agile cryptographic solutions for the cloud. *e & i Elektrotechnik und Informationstechnik* 134, 7 (nov 2017), 364–369. https://doi.org/10.1007/s00502-017-0519-x

[23] Gary McGraw. 2002. Building Secure Software: Better than Protecting Bad Software. *IEEE Software* 19, 6 (nov 2002), 57–58. https://doi.org/10.1109/MS.2002.1049391

[24] G. Mcgraw. 2004. Software security. *IEEE Security & Privacy Magazine* 2, 2 (mar 2004), 80–83. https://doi.org/10.1109/MSECP.2004.1281254

[25] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17.* ACM Press, New York, New York, USA, 311–328. https://doi.org/10.1145/3133956.3134082 arXiv:1708.08759

[26] Claus Pahl and Pooyan Jamshidi. 2016. Microservices: A Systematic Mapping Study. (2016). https://doi.org/10.5220/0005785501370146

[27] Henrich C. Pöhls and Focke Höhne. 2011. The Role of Data Integrity in EU Digital Signature Legislation - Achieving Statutory Trust for Sanitizable Signature Schemes. In *Revised Selected Papers from the 7th International Workshop on Security and Trust Management (STM 2011) (LNCS)*, Vol. 7170. Springer, 175–192. http://dx.doi.org/10.1007/978-3-642-29963-6_13

[28] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2006. *Security Patterns - Integrating Security and Systems Engineering.* John Wiley & Sons, Ltd. West Sussex, England.

[29] Osmanbey Uzunkol and Mehmet SabÄśr Kiraz. 2018. Still wrong use of pairings in cryptography. *Appl. Math. Comput.* 333 (sep 2018), 467–479. https://doi.org/10.1016/J.AMC.2018.03.062

[30] Rodolfo Villarroel, Eduardo Fernández-Medina, and Mario Piattini. 2005. Secure information systems development - A survey and comparison. *Computers and Security* 24, 4 (jun 2005), 308–321. https://doi.org/10.1016/j.cose.2004.09.011