Informatics in Education, 2020, Vol. 19, No. 3, 425–454
© 2020 *Vilnius University, ETH Zürich*
DOI: 10.15388/infedu.2020.19

Integrating Formal Methods for Security in Software Security Education

Paolo MODESTI

Department of Computer Science and Information Systems, Teesside University
Middlesbrough, United Kingdom
e-mail: p.modesti@tees.ac.uk

Received: October 2019

Abstract. As the number of software vulnerabilities discovered increases, the industry is facing difficulties to find specialists to cover the vacancies for security software developers. Considering relevant teaching and learning theories, along with existing approaches in software security education, we present the pedagogic rationale and the concrete implementation of a course on security protocol development that integrates formal methods for security research into the teaching practice. A novelty of the framework is the adoption of a conceptual model aligned with the level of abstraction used for the symbolic (high-level) representation of cryptographic and communication primitives. This is aimed not only at improving skills in secure software development, but also at bridging the gap between the formal representation and the actual implementation, making formal methods and tools more accessible to students and practitioners.

Keywords: software security education, formal methods for security, programming abstractions, research-led teaching, constructivism.

1. Introduction

Designing robust and secure systems is of paramount importance to protect digital assets and limit the attack surface available to the adversary. Vulnerabilities can be exploited by attackers, for example, to gain access to confidential data or compromise the integrity of online systems. This can cause direct (e.g. data loss, interruption of operations) or indirect harm (e.g. damage of reputation, liabilities) that may impact negatively on organisations and individuals. Many experts agree that the root cause of vulnerabilities is incorrect software (Dark, Belcher *et al.*, 2015; Walden and Frank 2006) and they recommend to invest more resources on enhancing the design of secure systems rather than on fixing vulnerabilities in operational systems. Unsurprisingly, prevention is better than cure.

As the number of vulnerabilities discovered and reported increases, the lack of experts in secure software development becomes more evident. Under the CVE (Common

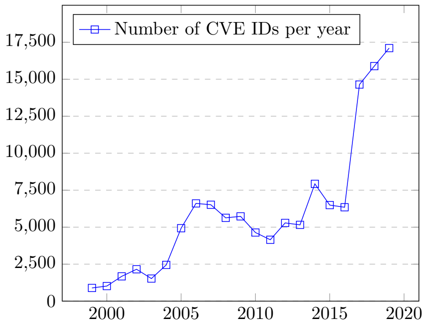


Fig. 1. Number of CVE IDs registered per year: 1999–2019 (Mitre 2020).

Vulnerabilities and Exposures) programme operated by MITRE, a unique identifier is assigned to publicly known vulnerabilities when they are reported. This repository is contributed by CVE Numbering Authorities (CNA), which along with MITRE includes major IT companies (e.g. Microsoft, Oracle, RedHat) and CERT Coordination Centres. The database does not include custom-built software. Fig. 1 displays the number of assigned CVE identifiers and a growing trend[1].

The (ISC)^2 Cybersecurity Workforce Study 2019 estimates that the current worldwide cybersecurity workforce gap is around 4.07 million ((ISC)2 2019). According to the Frost & Sullivan (2017) report for the Center for Cyber Safety and Education, the most high-value skills in short supply are: intrusion detection, secure software development and attack mitigation. A study of the Center for Strategic and International Studies (2016) in partnership with Intel Security also indicated secure software development as the second most scarce skill in cybersecurity.

The current situation represents, at the same time, a challenge and an opportunity for higher education institutions. Universities have to address a real need for businesses and society, and this allows us to propose academic programs offering a concrete career prospect in cybersecurity. In this regard, it is interesting to note that formal security education can allow new specialists to compete with more experienced professionals, in solving well-formalized but relatively new security tasks (Allodi *et al.*, 2018).

Focusing on software security, the following educational questions are relevant:

- How can higher education contribute to software security education?
- Which pedagogic approaches are available and on which teaching and learning theories can these approaches leverage?
- How to effectively teach the concepts and techniques employed for the design and implementation of secure systems?
- How can theoretical and applied research inform and lead the curriculum development?

[1] The sudden sharp increase in 2017 can be partially explained by the fact that more organisations have become CNAs and are now reporting vulnerabilities to the CVE repository. There were 48 CNAs at the end of 2016 and 110 in December 2019 (Mitre 2020).

Motivation and Contribution

In this paper, we try to answer these questions in a pragmatic way, taking into account relevant teaching and learning theories along with existing approaches to software security education. Notably, Mc-Gettrick (2013) advocated the need for the researchers working on secure systems development to inform the cybersecurity education community. Here, we propose a constructivist computer science education approach (Ben-Ari, 2001) in the context of software security, combined with research-led teaching (Griffiths, 2004), focusing on formal methods for security. In particular, we consider the symbolic modelling of cryptography (Dolev and Yao, 1983) and show how tools, built as research tools, can be applied in the teaching practice.

While previous works have considered challenges and solutions for secure software education from different perspectives (Section 2), we share ideas with authors that have considered a constructivist approach to teach, for example, networking protocols by programming in C++ (Pullen 2001) and modelling, evaluating and programming of secure systems with Java (Laschi and Riccioni 2008). Both papers propose support tools for the learning experience, however, our approach differs on two key aspects: the explicit use of the symbolic modelling of cryptography and the integration of tools and methodologies from formal methods for security research into the teaching practice. In particular, we propose a programming style that is aligned with the symbolic representation of high-level models, and we aim at bridging the gap between the formal representation and the actual implementation. In fact, while modelling and verification is a well established approach, how to derive a secure implementation from a model is not a trivial task (Bhargavan *et al.*, 2008).

As a proof-of-concept, we discuss the rationale for the design of a course on security protocols design and implementation (Section 4), based on a research-led and constructivist approach, and we propose a concrete implementation of such course (Section 5), applying tools and notions of formal methods for security. We also present a framework and the related workflow scenarios illustrating our approach as a whole (Section 7).

We focus on security protocols because they are an important building block for the construction of safe and robust applications as they play a key role in protecting data exchanged over a network infrastructure that can be assumed to be under adversary control, as in the Dolev-Yao attacker model (Dolev and Yao, 1983). Programming security protocols is challenging and notoriously error-prone; experience has shown that low-level implementation bugs are discovered even in protocols like TLS, SSH and WPA2 which are widely used and thoroughly tested. A significant example is the Heartbleed bug (Durumeric *et al.*, 2014) which is not a protocol failure, but rather a defect of the OpenSSL implementation of the TLS protocol.

To tame the complexity, the formal methods for security research community (Avalle, Pironti and Sisto, 2014; Bugliesi and Focardi, 2008) has advocated the specification of security protocols using high-level programming abstractions, suited for security analysis and automated verification. This was one of the main reasons for developing tools for verification of security protocols in the symbolic model (Blanchet *et al.*, 2019; Mödersheim and Viganò, 2009) and for the automatic generation of security protocols implementations (Almousa *et al.*, 2015; Avalle, Pironti, Pozza *et al.*, 2011; Modesti 2015).

Moreover, in order to widen the adoption of these tools among practitioners, simple languages based on the Alice and Bob notation (Mödersheim, 2009) have been adopted for the specification of security protocols (Basin *et al.*, 2015; Bugliesi, Calzavara *et al.*, 2016; Bugliesi and Modesti 2010). This simplifies the coding, especially for beginning or intermediate computer science students, and the code itself is more succinct than its equivalent in other formal languages like the process calculi (e.g. applied-pi calculus (Abadi and Fournet 2001)).

The main pedagogic goal of the course we propose is to teach, in a simple and effective way, how to build secure distributed applications using common cryptographic primitives (symmetric and asymmetric encryption, digital signature, hashing, message authentication codes) abstracting from their low-level details. The activities aim at helping students to quickly grasp the main security concepts and to effectively apply them to coding distributed programs. The constructivist approach allows students to put into practice what they are taught, building a viable mental model, and perceiving their progress working on increasingly more complex software artefacts.

Outline of the paper

Section 2 discusses approaches and strategies for software security education and Section 3 presents teaching and learning theories and considers their applicability to software security education. In Section 4, we discuss the rationale for the design of a course on programming security protocols and we introduce its structure and content in Section 5. The software tools used to support the learning activities and the framework are presented in Section 6 and 7 respectively. Finally, we discuss some lessons learned in Section 8 and conclude our presentation in Section 9.

2. Software Security Education

According to Schneider (2013) there are two rather different visions on how to develop university cybersecurity courses. The first one, in order to allow designer to see their systems from the same perspective attackers do, is to teach adversarial thinking. The second one is to focus on the principles and abstractions required to build secure systems. The latter approach is supported by authors like Bishop and Frincke (2005), Dark, Belcher *et al.* (2015), Pothamsetty (2005) and Walden and Frank (2006) advocating that academia and industry should focus on training software engineers to consider security in every aspect of the development process: requirements, design, implementation, testing and deployment. In particular, it is noted by Pothamsetty (2005) that, in many occasions, much more effort is put into symptoms (e.g., attack and defence techniques), than on the cause (defective design).

Jøsang *et al.* (2015) remark that many IT experts still have insufficient understanding of security and "it is irresponsible to offer IT programs at university without compulsory modules in information security".

Bishop and Frincke (2005) underline that adding security mechanisms to existing systems or fixing vulnerabilities is challenging as it requires a detailed understanding of many aspects of these systems. Moreover, they recommend that a key objective of an undergraduate computer science curriculum should be to enable students to write secure code.

Since in disciplines like computer science there is a significant applicative aspect, principles and practice are often blended. On the one hand, this meets the employers' expectation that graduates master both principles and practice. On the other hand, there is a concern (Bishop and Frincke, 2005) that the need for software security drives educators to teach specific programming languages, techniques and environments at the expenses of teaching concepts and principles that can be applied to new situations.

There is a tension between the need of learning how to solve current and specific problems and the ability to tackle new ones in the future. Some scholars like Bishop (2000) believe that undergraduate education should aim at teaching broad principles and their application, deliberately avoiding the focus on particular technologies, as the main objective should be to enable the understanding of general principles that can be abstracted and applied across many situations on different systems.

Other authors like Johnstone (2013) believe that a good understanding of how systems work in practice is unavoidable. They think this is true also in teaching secure coding to beginners, because, although some vulnerabilities can be mitigated with simple programming techniques, the most significant exploits are performed by individuals that have invested a considerable amount of time in understanding how software behaves and how to exploit it. Thus, to see if such vulnerabilities exist, students must understand how these exploits work.

Williams *et al.*, (2014) advocate that students should learn secure code principles along with different aspects of computer programming, but Johnstone (2013) reminds us of the concern of some instructors that teaching programming secure software systems can be another hindrance to learning coding. He agrees that students should learn how to follow secure coding practice to avoid writing insecure programs, but recognise that explaining how different exploits are performed may be difficult due to the lack of knowledge of how computer systems work.

Another question is how research can be applied to security education. Some authors like Bishop (2002) and Dark, Bishop *et al.*, (2015) observe that students are rarely exposed to research until they begin to work on their dissertation. They propose to offer a continuous exposure to scholarly work during the course of studies, in line with the *research-led teaching* approach (Griffiths, 2004). The objective is to apply research results and methodologies to real-world problems within a realistic applicative context. This approach helps students to bridge theory to practice and facilitates the assimilation of the latest techniques into their future work. However, from the educational perspective, the technical complexity of security research makes this task really challenging.

Along with the *research-led teaching* approaches, there are also other approaches (Yurcik and Doss, 2001): *tutorial-based approach* (McDermott and P. S. Shaffer, 1992), where the learner is guided through a series of activities, mostly a self-learning approach, that is often used to lead to the achievement of professional certifications, and the *project-based approach* (Bell 2010), which includes a project component in a course where students should demonstrate understanding of how to apply principles in practice. Additionally, Wen and Katt (2019) recently proposed a *context-based approach* in which knowledge is structured and presented by using an application scenario, stimulating learners' mental models and moving from concrete to abstract security knowledge.

All these approaches are not alternative, and they can be employed in a blended form to build academic and training courses.

3. Teaching and Learning Theories

In this section, we discuss teaching and learning theories that can be considered in software security education. When possible, we prefer to interpret teaching and learning theories, developed in a very general context, through the lens of previous scholar work focusing on the application of such theories to computer science and cybersecurity education.

Constructivism, and, to a lesser extent, cognitive load and behaviourism are learning theories that have attracted interest in computer science education (CSE) (Hadjerrouit, 2005; Quevedo-Torrero, 2009). To the best of our knowledge, there are few studies that focus explicitly on the application of such theories in teaching of software security. For example, Laschi and Riccioni (2008) and Pullen (2001) applied a constructivist approach to design their virtual labs, while Conklin and Dietrich (2007) considered elements of behaviourism. However, since much more works are available on teaching and learning theories in computer programming, it seems reasonable to consider their findings and observations also in the software security context.

- The *constructivist learning theory* (Bruner 2009; Wadsworth 1996) claims that knowledge is acquired by combining sensory data (gathered from experience) with the existing knowledge to create new cognitive structures. This process is applied recursively to generate new knowledge.
  Additionally, new knowledge is built reflecting on the existing one. To be effective, learning must be active, and the teacher must guide and facilitate students in their endeavour. The need to be active matches the everyday experience of computer science instructors: a passive learner is unlikely to develop the skills necessary to become a computer programmer (Hadjerrouit, 1998; Hadjerrouit, 2005; McConnell, 1996; Walker, 2004).
- The *cognitive load theory* (D. Shaffer *et al.*, 2003; Sweller, 2016) uses the knowledge of human cognitive processes to devise instructional procedures. It tries to define an information processing model to describe how the mind acquires and stores knowledge. It also creates a model that takes into account the limitations of the working memory. It has provided a theoretical framework for reasoning on the complexity of the learning process, useful, for example, to revise learning units and improve their effectiveness when teaching complex concepts. Applications in CSE include the evaluation of development environments used in programming courses (Mason *et al.*, 2015; Moons and Backer, 2013).

- The *behaviourist theory* (Skinner, 1954) studies how learning is affected by changes in the environment. The teacher is the dominant figure and the learner has little room for evaluation and reflection. The attempt of this theory to prove that behaviour could be predicted and controlled has been widely disputed (e.g. Chomsky 1959). However, the notion of positive reinforcement has some application in CSE, for example in the tutorial-based approach (McDermott and P. S. Shaffer 1992), where a guided series of tasks and exercises is proposed to the learner.

According to Ben-Ari (2001), constructivist practices in CSE should allow students to "discover knowledge by themselves when placed in the appropriate situation". This implies the construction of viable mental models, "generating knowledge by combining the experiential world with existing cognitive structures". However, the absence of mental models is a critical reason why students face difficulties in learning coding. Misconceptions occur when students adapt and increase their knowledge frameworks to assimilate new concepts (Eckerdal *et al.*, 2006). Previous programming experience and the overloading of language, taking notions from one context and using it in another one, have been considered reasons for misconceptions (Clancy, 2004).

Authors promoting a constructivist perspective (Ben-Ari, 2001; Hadjerrouit, 1998) indicate the importance of understanding both abstract and concrete concepts. Ben-Ari (2001) underlines that application of constructivism must consider that "a (beginning) computer science student has no effective model of a computer", and that "the computer forms an accessible ontological reality". The latter concept manifests itself when students interacting with a computer can immediately realise the effect of the application of their mental models. In other words, the results of misconceptions are discovered instantly. Rightly, Ben-Ari (2001) notes that "there is not much point negotiating models of the syntax or semantics of a programming language". Tools used by students, like compilers used in programming activities, have implications for building mental models. In fact, "mental processes, tool use, and interaction with the world are regarded to be tightly bound together" (Knobelsdorf, 2015).

A consequence of this rigidity is that the model of the system (for example, how a computer works) must be taught explicitly (Ben-Ari and Yeshno, 2006). This raises the question of how detailed the model must be, and how and when to use abstractions. For example, Object-Oriented Programming (OOP) allows us to build complex software abstracting many details of the underlying system. However, if learners have no knowledge of the underlying system, how can they understand the abstractions? From a constructivist perspective, if models are not viable, they will make further learning difficult.

There is not a general consensus on when abstractions need to be introduced. Adams (1996) opposes the idea of postponing teaching OOP until late in the course of studies because at that time it is difficult to have an impact on the learners' low-level model, but he also believes that OOP should not be taught too early when students are not mature enough to assimilate properly the related concepts. Ben-Ari (2001) remarks that "advocates of an objects-first approach seem to be rejecting Piaget's view that abstraction (or accommodation) follows assimilation". In fact, practitioners who use abstractions usually have a fairly good knowledge of the underlying model.

Conklin and Dietrich (2007) propose a paradigm to secure software engineering, which includes also elements of behaviourist theories. They underline that the knowledge of primitives and building blocks is essential and that these core concepts must be taught along with the system concepts. To facilitate the acquisition of core elements, they propose a series of experiential repetitive activities, of increasing level of difficulty, with system level issues introduced later when the student are already familiar with the primitive elements. As the level of difficulty rises, students need to adapt and improve their output (Bishop and Orvis, 2006).

Although Conklin and Dietrich (2007) do not mention it explicitly, in their framework they apply the behaviourist notion of positive reinforcement (Skinner, 1954). The authors also refer to Bloom's theory (Bloom and Krathwohl, 1956) which states that education should focus not on simply transferring notions, but rather on mastering the subject and encourage reasoning more conceptually. Conklin and Dietrich (2007) consider the primitives-based material as a knowledge type learning task, "based on rote memorization or recall of information", while systems level material is assimilated using "a combination of comprehension and application learning styles".

4. Rationale for the Design of a Course on Security Protocols Design and Implementation

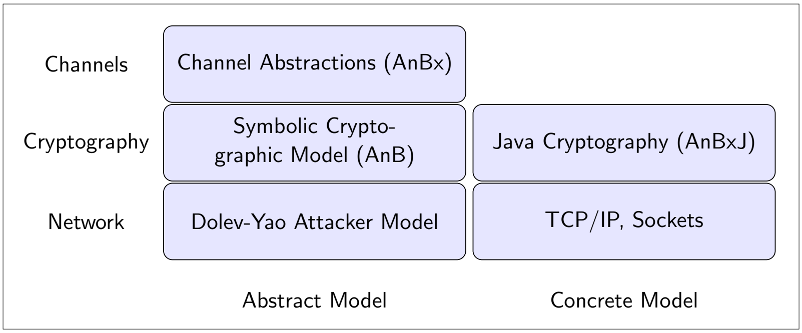
Leveraging on the background work presented in the previous sections, we can now discuss the rationale for the design of a course on programming security protocols. They are relevant as they play a key role in protecting data exchanged over a network infrastructure that can be under adversary control, and, as we have seen, programming security protocols is challenging and error-prone.

The main pedagogic goal of the course is to teach, in a simple and effective way, how to build secure distributed applications using common cryptographic primitives (symmetric and asymmetric encryption, digital signature, hashing, message authentication codes) abstracting from their low-level details. This course is aimed at helping students to quickly grasp the main security notions and to effectively apply them to the construction of distributed programs that can guarantee security goals like authentication and confidentiality.

It should be noted that our objective is not a mere transfer of knowledge but we aim at obtaining a cognitive change in the learner. Therefore, the course is also oriented at getting the students familiar with techniques like formal modelling and verification of security protocols and, when time allows it, also Model Driven Development (MDD) (Atkinson and Kuhne, 2003).

For the construction of this course, we identified the needs to be addressed as a series of design and pedagogic questions, inspired by the constructivist approach presented in Ben-Ari (2001).

We consider mainly constructivism here, for the reasons exposed in Section 3, in particular in the works of Ben-Ari (2001; 2006) and Hadjerrouit (1998; 1999; 2005). In summary, constructivism allows for the construction of a viable mental model, and help misconceptions to be discovered soon. Knowledge is actively built by learners in-



	Abstract Model	Concrete Model
Channels	Channel Abstractions (AnBx)	
Cryptography	Symbolic Cryptographic Model (AnB)	Java Cryptography (AnBxJ)
Network	Dolev-Yao Attacker Model	TCP/IP, Sockets

Fig. 2. Abstract and Concrete Model.

teracting with the environment and software tools contribute in shaping the way mental models are built. In fact, in constructivism, mental processes, tools and interaction with the world are tightly bound together (Ben-Ari, 2001).

To this end, the choice of tools, along with the pedagogic reasons, is crucial in this context. However, this choice can be flexible, as in general, teachers can consider their own preferences and interests in term of software tools for modelling and verification, and specification/programming languages, as long as the choice of tools is coherent and consistent with the approach and the learning objectives. Therefore, the concrete instance of the course we propose should be seen as a proof-of-concept, inspired also by our research interests and practice. Therefore, when necessary, we briefly introduce here the tools that will be presented in more details in Section 6.

Our conceptual framework is represented in Fig. 2, showing the abstract and concrete models, for the three different layers: Network, Cryptography and Channels.

What is the appropriate model of the "system"?

First of all, from the constructivist point of view (Ben-Ari, 2001; Hadjerrouit, 1998), we need to consider what is the appropriate model of the "system". In the domain of security protocols, there are two main approaches: computational and symbolic (Blanchet, 2012). In the computational model, messages are bitstrings, and cryptographic primitives are functions mapping bitstrings to bitstrings. The model considers the computational properties of the cryptography primitives (e.g. key size) and the adversary is any probabilistic Turing machine.

In the symbolic model, like the Dolev and Yao (1983) adversary model, the cryptographic primitives are represented by function symbols, assuming perfect cryptography. The adversary is restricted to use only such primitives. Fig. 3 shows the intruder rules, and the fact $\mathsf{iknows}(m)$ denotes that the intruder knows the term m. Every communication between honest agents is assumed to be mediated by the intruder, i.e., it happens through $\mathsf{iknows}(\cdot)$ facts. The model assumes the existence of a set of function symbols (with an associated arity) partitioned into two subsets of *public* and *private* symbols. The first rule describes both asymmetric encryption and signing ($\{\cdot\}$), while the second one models that a ciphertext can be decrypted if the corresponding decryption key is known. $\mathsf{inv}(\cdot)$ is a *private* function symbol representing the secret component of

$\mathsf{iknows}(M).\mathsf{iknows}(K)$	$\Rightarrow$	$\mathsf{iknows}(\{M\}_K)$	*Asymmetric Encryption*
$\mathsf{iknows}(\{M\}_K).\mathsf{iknows}(\mathsf{inv}(K))$	$\Rightarrow$	$\mathsf{iknows}(M)$	
$\mathsf{iknows}(\{M\}_{\mathsf{inv}(K)})$	$\Rightarrow$	$\mathsf{iknows}(M)$	
$\mathsf{iknows}(M).\mathsf{iknows}(K)$	$\Rightarrow$	$\mathsf{iknows}(\{\!\vert M\vert\!\}_K)$	*Symmetric Encryption*
$\mathsf{iknows}(\{\!\vert M\vert\!\}_K).\mathsf{iknows}(K)$	$\Rightarrow$	$\mathsf{iknows}(M)$	
$\mathsf{iknows}(M).\mathsf{iknows}(N)$	$\Rightarrow$	$\mathsf{iknows}(M,N)$	*Tupling*
$\mathsf{iknows}(M,N)$	$\Rightarrow$	$\mathsf{iknows}(M).\mathsf{iknows}(N)$	*Projection*
$\mathsf{iknows}(M_1).\cdots.\mathsf{iknows}(M_n)$	$\Rightarrow$	$\mathsf{iknows}(f(M_1,\ldots,M_n))$	*Function Application*

Fig. 3. Dolev-Yao intruder rules.

a given key-pair. The third rule allows the attacker to learn the payload of any known signed message. Symmetric encryption ($\{\!|\cdot|\!\}$) can be modelled similarly to the first two rules but in this case the same key is employed for both encryption and decryption. Additionally, there are rules for tupling and projecting tuple elements, and a rule for the application of *public* function symbols to known messages. Constants, including agent identities, are modelled as public functions with zero-arity.

We consider, for the scope of this work, the symbolic model being more appropriate than the computational one. In fact, Haberman and Kolikant (2001) found that a black-box-based approach can be used effectively to introduce programming concepts to novices. An important characteristic of the symbolic model is its simplicity. According to the constructivist approach, the model must be taught explicitly (Ben-Ari and Yeshno 2006; Koppelman and van Dijk, 2010), therefore a simpler model is of great advantage.

Interestingly, the model is also realistic. In fact, Herzog (2005) proved that there are many significant cases in which the Dolev-Yao adversary can be a valid abstraction of all realistic adversaries. Moreover, the model is amenable for automated verification of security protocols (Blanchet, 2001; Mödersheim and Viganò, 2009; Schmidt *et al.*, 2012) and this is important, not only in practical terms, but also pedagogically, to address the question of accessing the “ontological reality” discussed later.

What is the suitable level of abstraction?

Along with the adversary model, we need to teach how to model cryptographic primitives in the symbolic model, and their properties. This is necessary to allow the learner to understand the actions that honest agents perform during the protocol execution. This is required to understand the model and define a level of abstraction upon which the learners can build their knowledge. At this point, we should recall the recommendation given by Ben-Ari (2001) regarding the need to explicitly presenting a viable model one level beneath the one we are teaching. To this end, we propose a series of learning activities, aimed at the construction, in a real programming language (Java in our case), of simple security protocols.

The activities have an increasing level of difficulty in bite size steps, to help students to manage the complexity of the material. The purpose of these activities is also to elicit the prior knowledge needed to construct a viable model. In order to tame the complexity and abstract from low-level cryptography details, we rely on a Java library (Modesti, 2015) that offers an application programming interface (API) simplifying the access to a set of standard network and cryptographic primitives required to build and run security protocols implemented in Java.

The rationale of this approach is corroborated by Sivilotti and Lang (2010) who suggest to separate the concerns of the interface from the ones of the implementation. One potential challenge, as outlined by Alexandron *et al.* (2012), is that some programmers might perceive the right level of abstraction as the one that matches their programming experience. This could lead to a cognitive dissonance, that can "even lead to a negative attitude towards the high-level abstraction".

What is a language suitable for the specification of security protocols?

We propose the Alice and Bob language (AnB) (Mödersheim, 2009), a simple and intuitive notation, that describes messages exchanged by different agents, and allow for the formal specification of security goals in a human readable format. Symbolic cryptographic functions can be used to compose messages. This notation makes the coding of security protocols considerably simpler (and more compact) than their equivalent in other formal languages (e.g. process calculi (Abadi and Fournet, 2001; Abadi and Gordon, 1997)) or real-world programming languages. This intuitive language allows learners to build their own models of security protocols and experiment with them using tools that support such notation like the model checker OFMC (Mödersheim and Viganò, 2009) and Tamarin (Schmidt *et al.*, 2012).

```
Protocol: Fresh_From_A AnB

Types:
    Agent A,B;
    Number Msg,Nonce;
    Function [Agent -> PublicKey] pk,sk

Knowledge:
    A: A,B,pk,sk,inv(pk(A)),inv(sk(A));
    B: B,pk,sk

Actions:
    A -> B: A
    B -> A: {Nonce,B}pk(A)
    A -> B: {Nonce,B,Msg}inv(sk(A))

Goals:
    B authenticates A on Msg
    inv(pk(A)) secret between A
    inv(sk(A)) secret between A
```

Fig. 4. AnB Protocol Example.

An example of an AnB protocol is displayed in Fig. 4. The main goal of the protocol is to achieve authentication (precisely the injective agreement defined by Lowe (1997)) of the message `Msg`. The recipient `B` should have evidence that the message has been endorsed by `A` and is a fresh message. The goal is achieved using asymmetric encryption and a challenge-response technique with a `Nonce` exchange. `pk()` and `sk()` are abstract functions used to model asymmetric encryption, mapping agents to their public keys, for encryption and signature respectively, while `inv()` is a private function modelling the private key (its argument is a public key). It should be noted that private function symbols are used to represent symbolically a notion, they are not concrete functions that can be computed (Mödersheim, 2009).

Can we further abstract?

Abstracting from low-level details, where most implementation errors occur (Tsipenyuk *et al.*, 2005), the developer can focus on the application design and its security properties. For this reason, the formal methods research community (Abadi, Fournet and Gonthier, 2002; Avalle, Pironti and Sisto, 2014; Bugliesi and Focardi, 2008) have advocated the specification of security protocols with high-level programming abstractions, suited for security analysis and automated verification.

Concretely, we can abstract from cryptographic details, using the AnBx language (Bugliesi, Calzavara *et al.*, 2016), an extension of the AnB language. In AnBx, channels are the main abstraction for communication, providing different authentication and/or confidentiality guarantees for message transmission. As we try to build further knowledge on top of the existing one, we may help students to build viable models at an higher level of abstraction. For example, if we consider the example in Fig. 4, the three actions could be replaced by a single action `A->B,(A|B|-): Msg` where `(A|B|-)` represents a channel that allows to send a message authentic from `A` that can be verified by `B`. In this case, the actual implementation of the channel can be abstracted (provided it satisfies the desired goal).

Alexandron *et al.* (2014) suggest that when programmers can work with a less detailed mental model, it becomes easier to work with high-level abstractions. This might work also in our case, since this will reduce the gap between the level of the problem (i.e. security goals) and the level of the implementation.

How to access the "ontological reality" in this framework?

Learners can model protocols and reason about their security properties using tools for the verification of security protocols in the symbolic model. In a nutshell, the student can specify the security protocol and its security goals in AnB and then verify whether the protocol satisfies these goals, or if an attack may occur (in this case an attack trace is provided).

It should be noted that, with the specification of security goals (section `Goals` in Fig. 4), the learners describe the expectation regarding the security properties of the protocol, that reflects their mental model. Running the verification tools, provides an (almost) immediate feedback on the correctness of the mental model and helps to build their own knowledge autonomously. The analysis of errors is therefore an opportunity for individual reflection.

Moreover, as Ben-Ari (2001) advises, “when a student makes a mistake or otherwise displays a lack of understanding, you must assume that the student has a more-or-less consistent, but non-viable, mental model”. The task of the teacher “is to elicit this model and guide the student in its modification”.

Compiling and running programs in Java, as well as model checking the abstract model of a security protocols, also gives the student the opportunity to reflect on the error messages and on the run-time behaviour of the programs. This is in line with the principle advocated by Lebow (1993), and applied in practice for the design of the Network Workbench (Pullen 2001), of strengthening “the learner's tendency to engage in intentional learning processes, especially by encouraging the strategic exploration of errors”.

5. Course Structure

As a proof-of-concept, based on the principles discussed in the previous sections, we now outline the structure of a course in which learners need to develop, within very tight time constraints, their ability to implement secure communication systems, identifying and evaluating the security aspects required at different stages of the system development life-cycle. A particular focus is given to the role of applied cryptography in secure systems design and implementation. Therefore, students are introduced to the application of cryptography and specifically on those cryptographic primitives utilised in security protocols design and implementation.

In detail, the course addresses the following:

- **Learning needs:** as the students may come from different degree programs (e.g. computer science, computing, cybersecurity), their programming skills may be mixed. Therefore, the traditional learning curve to master cryptography is likely to be too steep for many students. No prior knowledge of network or cryptography programming is assumed.
- **Purpose:** in a limited amount of time (e.g. 5 hours of lectures, 8–10 hours of supervised tutorials, and 15–20 hours of independent work/self-study, but shorter instances are also possible) students should be able to gain an understanding of secure design principles, in particular demonstrating the ability to implement simple security protocols applying cryptographic primitives like symmetric and asymmetric encryption, digital signatures, message digests and message authentication codes.
- **Activities:** while the lecture sessions focus on the theories and the challenges associated with putting them into practice, the lab-based tutorial sessions help the student to manage the increasing complexity of the material being delivered. Activities are detailed in §5.4.
- **Learning outcomes:** critical understanding of principles of secure design, security goals and attacker capabilities; ability to implement in reliable and effective way simple security protocols, applying cryptography and defensive programming.

5.1. *Scope*

With respect to the ACM, IEEE, AIS & IFIP" Cybersecurity Curriculum 2017" (Joint Task Force on Cybersecurity Education, 2017), the scope of this course falls within the *Data Security* and *Software Security* knowledge areas (KA). Clearly, other cybersecurity areas like, for example, ethical hacking and information security auditing, stay out of the scope. Therefore, if they are included in the curriculum of a degree program, they shall be covered by other learning activities.

5.2. *Prerequisites*

We consider now the prerequisites for this course with reference to the "Computer Science Curricula 2013", a document covering curriculum guidelines for undergraduate degree programs in computer science written by the ACM and IEEE Computer Society Joint Task Force on Computing Curricula (2013).

The most important prerequisite is a sufficient theoretical and practical knowledge of programming. In our case, we expect most students having learned Java, C++ or C# in the past, at a level that can be described as at least equivalent to the *Object-Oriented Programming* unit (Core-Tier1) of the *Programming Language* (PL) KA of the "Computer Science Curricula 2013". This includes object-oriented design, class-hierarchy design for modelling, definition of classes (fields, methods, and constructors), subclasses, inheritance, and method overriding. Knowledge in this area implies *Software Development Fundamentals* (SDF) Core-Tier1, e.g. Algorithms and Design, Fundamental Programming Concepts, Fundamental Data Structures.

The security prerequisites include the foundational concepts of security (Core-Tier1) as defined in the *Information Assurance and Security* (IAS) KA. For example, CIA (confidentiality, integrity, availability), risk, threats, vulnerabilities, attack vectors, authentication and authorization, access control, trust and trustworthiness.

For the networking prerequisites, we refer to the *Networking and Communication* (NC) KA: prior knowledge shall include the organization of the Internet, basic knowledge of internet protocols, networked and distributed applications (e.g. client-server model).

No prior knowledge of network and cryptography programming is assumed, and the basic knowledge of cryptographic primitives is acquired during the lectures that are part of the course. While further prior knowledge of Information and Network Security can be helpful, our experience (see Section 8) suggests that overall these prerequisites are sufficient for the objectives of this course.

5.3. *Content*

The course is structured as a series of lectures, each followed by a practical tutorial that not only allows to apply and reinforce what learned during the lecture, but also stimulate further learning. The main topics covered are:

- Principles of secure design.
- Security protocols and security goals.
- Client-Server programming.
- Key generation.
- Public Key Infrastructure (PKI): setting up a certification authority, registration and certification process.
- Simple secret exchange, based on asymmetric encryption.
- Weak authentication using digital signature.
- Strong authentication using challenge-response and digital signature.
- Implementation of a secure channel combining secrecy and authentication.
- Defensive programming/secure coding: checks on reception.
- Configuration of cryptographic algorithms.
- Modelling and verification of security protocols specified in the AnB/AnBx.
- Model Driven Development (MDD) of security protocols.

The last two topics can be omitted or briefly mentioned if the course runs in the shorter variant (basically limited to the programming part).

5.4. *Activities*

The practical programming tutorials[2] are based on the Java programming language and the cryptographic services offered by the Java Cryptography Architecture (JCA). We chose this language and toolkit because they are widely adopted by the industry and freely available. Moreover, we use the AnBxJ library which is part of the AnBx Compiler and Code Generator (Modesti, 2015), an open-source tool developed by the author as an academic research project aimed at the automatic generation of security protocol implementations, from a simple, formally verifiable, abstract model. The AnBxJ library wraps, in an abstract way, the JCA interface and implements the custom classes necessary to write programs in Java. This allows to escape the complexity of the JCA programming interface, and offer to the developer a simplified API for cryptography and communication. For the modelling and verification phase, we employ the OFMC model checker (Mödersheim and Viganò, 2009), a tool for symbolic security protocol analysis that supports the Alice and Bob notation. The hierarchy of tools and notions, used in this course, for abstract and concrete models is shown in Fig. 2.

Fig. 5 shows an example of the kind of Java code students must be able to understand and write. Since we map our symbolic model to concrete primitives that preserve the syntactical simplicity of the abstract one, there is no need, even in the concrete model, to explicitly use low-level network and cryptographic primitives (e.g. sockets and ciphers). Therefore, the exchange of messages can be done by simply invoking the **`send()`** and **`receive()`** methods, while **`encrypt()`**, **`decrypt()`**, **`sign()`** and **`verify()`** can be used to implement public key encryption and digital signature.

[2] Available at `https://paolo.science/anbxtutorial/`

```
import anbxj.*;
import java.security.SignedObject;

class Server_Fresh_From_A {

public Server_Fresh_From_A (AnB_Crypto_Wrapper anb, Channel_Properties cp) {
  String name;
  Channel_Abstraction s = Channel.setup(cp);
  do {
    // the server waits for incoming connections
    s.Open();
    String msg = new String("Hello! What's your name?");
    s.Send(msg);
    // A -> B: A  # A: client, B: server
    name = (String) s.Receive();
    // B -> A: {Nonce,B}pk(A)
    Crypto_ByteArray nonce = anb.getNonce();
    msg = "Hi " + name + "! Please sign this challenge and your message:\n"
         + anb.getName() + "," + nonce.toString()
         + "\nCut&Paste and press <ENTER>"
    s.Send(anb.encrypt(msg,name));
    // A -> B: {Nonce,B,Msg}inv(sk(A))
    msg = (String) anb.verify((SignedObject) s.Receive(),name);
    if (msg.equals(null))
      s.Send("Sorry " + name + ", I am unable to verify your signature");
    else
      // we omit here the checks on nonce and agent's name
      s.Send("Thank you " + name + ", your signature has been verified\n"
      + "Your message was: " + msg);
    System.out.println(msg);
    s.Close();
  } while (true);
  }
}
```

Fig. 5. Java Protocol Example – Server class

It should be noted that the code in Fig. 5 is, conceptually, very similar to the AnB model (Fig. 4). This is possible because the details of the actual code implementing the networking and cryptographic functionalities is shielded by the AnBxJ library (see §6.1). Therefore, the code that student use/write preserves the high-level and declarative nature of AnB. According to Alexandron *et al.* (2014) and Haberman (2004) this approach can reduce to the cognitive load which is in general introduced by the complexity of the OOP abstractions.

5.4.1. *Examples*

The tutorials are structured in a series of activities of increasing complexity. Each programming task usually starts with the source code of a program that students have to analyse and run. As the AnBxJ library allows running the application with different debugging levels, log messages are available to report about every step of the programs execution. Students are then asked to modify or extend the program in order to add some functionalities or to achieve some security goals, thus applying and reinforcing their knowledge. Examples of activities include:

1. **Client/Server:** the sample program runs a client and a server on the same machine. Students are asked to modify the program in order to allow client and server to run on two different machines. Next, the application should be amended to swap the role of client and server.
2. **Key generation and usage of a PKI:** the `keytool` command (available within the Java JDK) is used to generate cryptographic keys and save them into keystores. `cfssl` (CloudFlare's PKI/TLS tool) is used to set up a certification Authority and for the registration and certification process.
3. **Secret exchange:** the sample program performs a simple secret exchange between two agents using asymmetric encryption. Cryptographic keys for different agents are provided. Students are required to run the program using different identities. The output of the program is analysed in order to identify successful and unsuccessful run of the program. Then they are asked to modify the code in order to allow different agents to perform different roles (client or server).
4. **Authenticated exchange:** an activity similar to the previous one, but focusing on the authentication mechanism (considering digital identity and digital signature). Students are also asked to implement the checks on reception, actions performed on the receiver side in order to verify if the incoming messages meet the ones expected according to the protocol specification. Therefore, this is also an exercise on secure coding/defensive programming.
5. **Secure channel:** provided that in the previous tasks students have successfully used encryption and digital signature, they are asked to implement a secure channel combining the two techniques.
6. **Modelling and verification of security protocols:** students use the model checker (OFMC in our case) to test a set of given protocols in order to verify if they satisfy the expected security goals. Afterwards, they are required to modify the safe protocols in order to violate some security goals and explain why protocols fail. Conversely, unsafe protocols will need to be fixed.
7. **(Manual) implementation of security protocols:** from a formally verified model, students are asked to write the code that implements the protocol in Java. Previous examples can be used as a template.
8. **Model driven development:** students can now automatically generate a Java application from a model formally verified, using the AnBx compiler. They can compare their solutions to the previous exercises with the code automatically generated. It should be noted that the AnBx compiler automatically generates the checks on reception, giving the opportunity to compare these checks with the ones encoded by hand in the previous exercises.

6. Toolkit

Although, as previously discussed, the actual choice of tools and languages for the practical activities can be influenced by the instructor's preferences and interests, we think it is useful to give a concrete example of a toolkit and describe possible workflow scenarios (Fig. 8 and Fig. 9) that can support the learning activities. Therefore, we briefly describe here the tools that, along with the Java programming language and development kit (JDK), are part of the toolkit we use for the practical tutorials. These tools were developed as part of academic research projects in the area of formal methods for security. In fact, we believe it is important, in the spirit of research-led teaching, to expose students not only to practitioner's tools, but also to state-of-the art research tools than can be applicable in the professional practice.

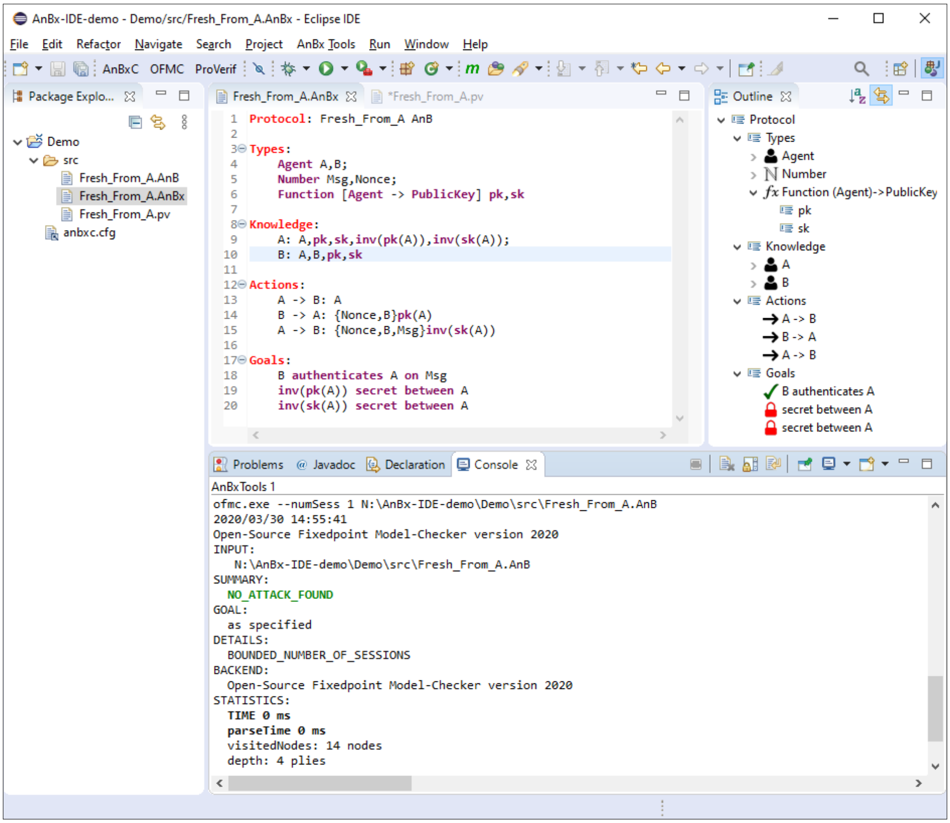
All tools are accessible from the command line and are available under Windows, Mac and Linux. It is also possible to use an Eclipse plug-in (AnBx IDE (Garcia and Modesti, 2017)) that extends the standard Java programming support offered by the Eclipse IDE with functionalities for the design, verification and implementation of security protocols that can support the learning activities. Along with providing a GUI access (Fig. 6) to the toolkit, the plug-in simplifies the setup and utilisation of the tools.

6.1. *AnBxJ Library*

This Java library provides an application programming interface (API) that implements the communication and cryptographic primitives required to run the programs. To provide exibility, the API does not commit to any specific cryptographic solution (algorithms, libraries, providers). Instead, it is structured as a modular, easily configurable, framework that leaves the developer free (at compile, deployment or even at runtime) to decide which cryptographic scheme to use, according to the cryptographic strength and performance requirements the application must satisfy.

The API is structured as a layered architecture (Fig. 7), whose main components are:

- The *transport layer* provides all the networking functionality necessary to transmit messages over the network, using both plain and secure sockets (TLS). In fact, although the enforcement of the security properties is often delegated to the cryptographic layer, it is also possible to run applications over a secured channel rather than over a plain one.
- The *cryptographic layer* essentially provides methods to encrypt and decrypt, sign and verify, digest messages using the features included in libraries like **`java.security`** and **`javax.crypto`**. The public key infrastructure (PKI) binds public keys with their respective user identities by means of a certificate authority (CA).



```
Protocol: Fresh_From_A AnB

Types:
    Agent A,B;
    Number Msg,Nonce;
    Function [Agent -> PublicKey] pk,sk

Knowledge:
    A: A,pk,sk,inv(pk(A)),inv(sk(A));
    B: A,B,pk,sk

Actions:
    A -> B: A
    B -> A: {Nonce,B}pk(A)
    A -> B: {Nonce,B,Msg}inv(sk(A))

Goals:
    B authenticates A on Msg
    inv(pk(A)) secret between A
    inv(sk(A)) secret between A
```

```
AnBxTools 1
ofmc.exe --numSess 1 N:\AnBx-IDE-demo\Demo\src\Fresh_From_A.AnB
2020/03/30 14:55:41
Open-Source Fixedpoint Model-Checker version 2020
INPUT:
   N:\AnBx-IDE-demo\Demo\src\Fresh_From_A.AnB
SUMMARY:
  NO_ATTACK_FOUND
GOAL:
  as specified
DETAILS:
  BOUNDED_NUMBER_OF_SESSIONS
BACKEND:
  Open-Source Fixedpoint Model-Checker version 2020
STATISTICS:
  TIME 0 ms
  parseTime 0 ms
  visitedNodes: 14 nodes
  depth: 4 plies
```

Fig. 6. Toolkit GUI.

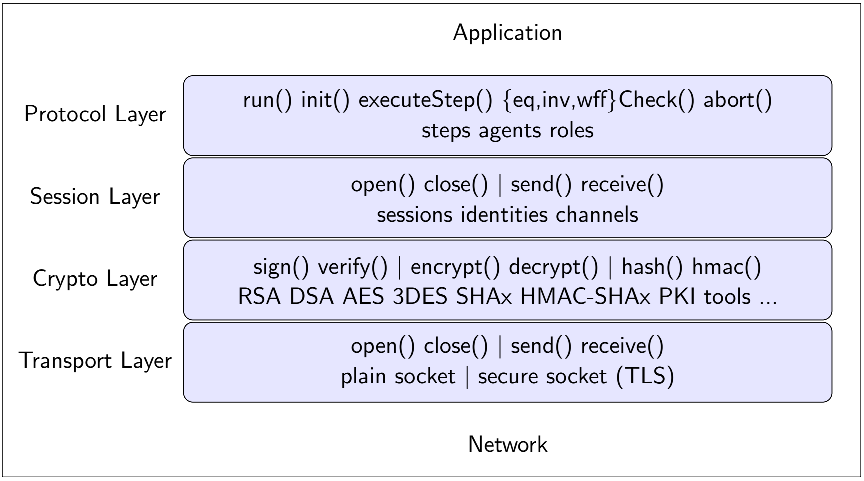

Application
Protocol Layer
run() init() executeStep() {eq,inv,wff}Check() abort()
steps agents roles
Session Layer
open() close() | send() receive()
sessions identities channels
Crypto Layer
sign() verify() | encrypt() decrypt() | hash() hmac()
RSA DSA AES 3DES SHAx HMAC-SHAx PKI tools ...
Transport Layer
open() close() | send() receive()
plain socket | secure socket (TLS)
Network

Fig. 7. AnBxJ Java Library Architecture.

- The *session layer offer*s the functions to **send()** and **receive()** data. Any *serializable* object can be a *message* exchanged by means of these primitives, thus it is possible to transmit a wide range of object classes across a network connection link. Methods to **open()** and **close()** sessions are also provided.
- The *protocol layer* gives an abstract description of the protocol: data and control ow, steps and principal roles, and checks on reception.

6.2. *OFMC Model Checker*

For the tutorials, the OFMC model-checker (Mödersheim and Viganò, 2009) is used for the verification of abstract models. Students can experiment with the specification of protocols and the verification of their security goals. OFMC employs the AVISPA Intermediate Format IF (AVISPA, 2003) as "native" input language, defining security protocols as an infinite-state transition system using set-rewriting. Notably, OFMC also supports the more intuitive language AnB (Mödersheim, 2009), and the AnB specifications are automatically translated to IF.

OFMC performs both protocol falsification and bounded session verification by exploring, in a demand-driven way, the transition system. If a security goal is violated, an attack trace is provided.

6.3. *AnBx Compiler and Code Generator*

The AnBx Compiler and Code Generator (Modesti, 2015) is an automatic Java code generator for security protocols specified in AnBx or AnB. In the tutorials, AnBx is used in the context of Model Driven Development. Provided that a model has been validated with OFMC, the user can automatically generate a Java implementation. This is useful to familiarise with the software engineering approach of Model Driven Development, but also to compare a manual implementation with an automated one. The main features of the compiler are:

- Automatic computation of the defensive checks that an agent has to perform on incoming messages.
- Optimisation of cryptographic operations in order to minimise the number of computational steps and reduce the overall execution time (Modesti, 2014).
- Mapping of abstract types and API calls to the concrete ones provided by the AnBxJ library.
- A set of template files is used to generate the code. Template files can be customised, for example, to integrate the generated application in larger systems.

Since the compiler translates the intermediate format to the Applied pi-calculus (Blanchet 2001), the verification of the protocol logic used for the code emission phase can be performed with the protocol verifier ProVerif (Blanchet *et al.*, 2019).

7. Framework and Workflow Scenarios

An overview of the framework is shown in Fig. 8, and is inspired by the AnBx Compiler architecture (Modesti, 2015). The main ideas behind this framework have been discussed in Section 4 when we presented the rationale behind the course. The toolkit presented here can be employed in a variety of configurations. In particular, we identify four main workflow scenarios related to the learning activities (Fig. 9):

1. Activities involving the analysis, editing and execution of existing code or writing new code from an informal description of the application and its requirements. Tools involved are mainly the JDK and the AnBxJ security library. These activities allow to familiarize with the communication and cryptographic functions necessary to build secure distributed applications. Tasks include, for example, Client/Server programming, key generation and setup of a PKI, construction of distributed applications implementing secret, authentic and secure channels. The execution of the application may require the editing of the configuration file which provides the runtime parameters to the underlying layers (JRE and JCA). Students are free to use different design approaches (e.g. pseudo-coding, flow-charts,

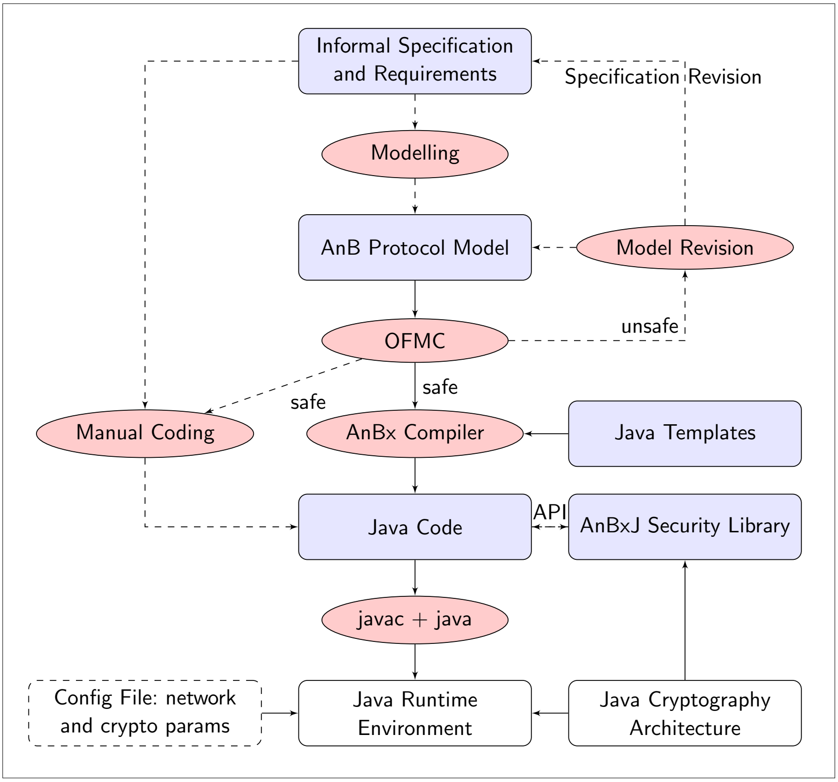


Fig. 8. Framework: Toolkit and Workflow (– – – manual —— automatic); adapted from (Modesti 2015).

#	*Workflow Scenario*	*JDK*	*AnBxJ*	*OFMC*	*AnBxC*	*Examples of activities* §5
1	Manual coding of program from the requirements or a given source code	✓	✓	–	–	1,2,3,4,5
2	Abstract modelling of the security protocol from the requirements, and verification (model checking)	–	–	✓	–	6
3	#2 + Manual coding of the program from the abstract model	✓	✓	✓	–	6,7
4	#2 + Automatic generation of the program from the abstract model (MDD)	✓	✓	✓	✓	6,8

Fig. 9. Workflow scenarios and related tools.

UML diagrams, etc.) but given the high-level of abstraction offered by the AnBxJ library, writing the code directly is a viable option as the level of abstraction is reflected in the programming style (see Fig. 5).

2. Activities involving the abstract modelling of security protocols from their requirements and/or informal description. In our case, for the verification task, the main tool used in these activities is the OFMC model checker. The tasks are aimed at getting the students familiar with abstract reasoning and formal modelling. The specification of the model involves the need to specify in AnB not only the actions performed by the agents, but also their initial knowledge. A crucial aspect in this phase is the specification of the security goals that protocols are meant to achieve. This is a critical and often neglected activity as most informal descriptions of protocol used in the industry fail to give a rigorous definition of the security goals but rather use the natural language to describe them. This is the case, for example, of the ISO/IEC 9798 standard for entity authentication (ISO/IEC 2010). Once the model is tested, unsafe protocols may require an iterative process of revision and verification of the model until the protocol is successfully verified. In some cases, even the requirements may need a revision if they are ambiguous or ill-formed. Although the students use the model checker as a black-box security oracle, the tool can print an attack trace if the protocol is unsafe. This provides elements that help understanding why the protocol fails in satisfying the security goals. Such iterative process is quite standard in formal design of security protocols, and it is aimed at capturing design errors in the very early phases of software development.
3. After modelling and verification (#2), manual coding of the Java application can be attempted. Practising this activity allows to develop the skills needed to bridge the gap between the abstract and the concrete model. It also integrates the knowl-

edge developed in the scenarios #1 and #2. This is a crucial step in the learning process as, like in software engineering, this is a phase where implementation errors typically occur, e.g. Durumeric *et al.* (2014). Therefore, an important focus is given to the implementation of the defensive checks on incoming messages to allow students to develop the adversarial thinking.

4. After modelling and verification (#2), a Java implementation can be generated automatically from the AnB abstract model with the AnBx compiler. This activity allows to become familiar with the Model Driven Development. Further activities involve the comparison of the manual implementation, in particular the defensive checks, with the automatically generated one. This scenario involves the usage of all tools of the framework.

8. Lessons Learned

Although the main purpose of this paper is to present the theoretical foundations of the framework and its pedagogic approach, we discuss here some lessons learned running two instances of this course at the University of Sunderland (UK), during the academic year 2017–2018. Given the limited data available, we do not claim that these results are decisive, as an evidence-based controlled experiment, part of a future work, will be necessary for it. Nevertheless, we consider useful to report some reflections on the teaching practice so far.

In both deliveries of the course, students did not have any prior knowledge in security programming and implementation of security protocols, but only some programming and general cybersecurity and networking notions (not including cryptography), in line with the prerequisites described in §5.2. For contingency reasons, this learning activity was part (as a short course) of the *Advanced Cybersecurity* module (undergraduate, year 3) and, in the complete version, of the *Principles of Cybersecurity and Cyber Resilience* module (postgraduate taught).

For the latter course, the development of a secure application was a component of the formal assessment. Students were asked to implement a secure PIN distribution system, where the communication channel between the client and the server should provide both secrecy and authentication. For this cohort ($n = 13$), we analysed the submitted artefacts and computed a score (range: 0–100, η = average, σ = standard deviation) linked to the completion of the implementation of the required security and communication goals.

#	Security/communication goal	η	σ
1	Establish a client/server communication channel	100.0	0.0
2	The server should be able to authenticate the client	84.6	36.1
3	Encrypted channel between client and server	90.0	24.2
4	The client should be able to decrypt the data received from the server	90.0	24.2
	Participants ($n = 13$)		

Although this gives limited possible evidence of the effectiveness of the proposed approach, as data was not collected as part of a controlled (quasi) experiment and the participant number is too small, observation has nevertheless provided several interesting and promising insights:

1. First of all, the major achievement was that students without any specific security programming background were indeed able to build successfully simple distributed applications implementing security goals like authentication and secrecy.
2. Another important aspect was that, for the first time, students approached a topic like modelling and verification of security protocols. Interestingly, this allowed to demystify the common misconception among students new to this topic, that using strong cryptography is sufficient to build secure protocols. Indeed, understanding that defects in the design of the protocols may allow the intruder to attack them without the need to break any cipher scheme, proved to be extremely formative and deeply appreciated by the students.
3. With this respect, the symbolic approach has helped to develop the adversarial thinking, as formal verification requires a precise characterisation of the intruder in the Dolev and Yao (1983) style (Fig. 3). To this end, having to explicitly define the security goals of the model and ensure that these goals are satisfied at every stage, was crucial to assimilate this concept.
4. Moreover, this experience offered the students the opportunity to become aware and familiarise with methodologies and techniques, i.e. formal methods, that are not only used in academic research but also are becoming increasingly adopted by the industry. This may be also useful to help students in pursuing further studies or increasing chances to find a qualified job in the software development and cybersecurity sector. Overall, the students appreciated that that security needs to be considered from the early stages of the software life cycle.
5. We realized that students with limited OOP experience sometimes faced more difficulties than others in completing practical programming tasks. This was not totally unexpected because programming skill was one of the prerequisites. Students familiar with C#, which was the main programming language taught at the university, generally adapted rather quickly to our technical toolkit based on Java.
6. Supervised tutorials were run in laboratory and we used a formative evaluation strategy with one to one feedback from the tutor. Ongoing testing/evaluation was also embedded in the programming activities, guiding students to progress through the materials, improving gradually their coding technique as the course progressed.

These preliminary findings can be interpreted in light with the literature reviewed in the first part of paper. Firstly, constructivism allows for the construction of a viable mental model, and help misconceptions to be discovered soon. In particular, knowledge is actively build by learners interacting with the environment and, as in constructivism mental processes, tools and interaction with the world are tightly bound together (Ben-Ari, 2001), this contribute in the way mental models are built.

Secondly, the symbolic model and its black-box abstractions can be employed effectively to introduce programming concepts (Haberman and Kolikant, 2001). A crucial characteristic of our symbolic model is its simplicity and, since constructivism requires the model to be taught explicitly (Ben-Ari and Yeshno, 2006; Koppelman and van Dijk, 2010), this is extremely useful.

Thirdly, abstracting from low-level details, where most implementation errors occur (Tsipenyuk *et al.*, 2005), the developer can focus on the application design and its security properties. An additional advantage is that, as highlighted by Avalle, Pironti and Sisto (2014) and Bugliesi and Focardi (2008), such high-level programming abstractions are suitable for security analysis and automated verification, therefore abstract and concrete model are clearly related.

9. Conclusion

Building on previous pedagogic and technical scholarly work, in this paper we devised an approach for the development of a course on design and implementation of security protocols. Such approach leverages on constructivism and research-led teaching, integrating formal methods for security tools into the teaching practice. We also proposed a framework aimed at making such tools and methodologies more accessible to students and practitioners.

We think that along with the theoretical contribution, it was also useful to report about our initial experience and lessons learned. As noted by Allodi *et al.* (2018), there is a relative small number of experimental studies on the effectiveness of different software security education approaches. For instance, Tabassum *et al.* (2017) investigated how an IDE can support secure coding with instant security warnings, detailed explanations, and auto-generated remediation code. Bishop, Dark *et al.* (2019) considered how programming clinics can impact in assimilating secure programming principles into practice, and Theisen *et al.* (2016) compared the delivery of courses on software security on campus and MOOCs (Massive Open Online Courses).

To the best of our knowledge, there is not yet an experimental investigation about the application of high-level abstractions and formal method security tools in the spirit of what we proposed here. For this reason, in terms of future work, it would be useful to run a controlled experiment applying an evaluation methodology similar to the one used by Allodi *et al.* (2018) and Mirkovic *et al.* (2015) to evaluate cybersecurity education interventions.

Moreover, a pedagogic question that would be interesting to explore is the opportunity of changing the order in which the content is delivered, considering advantages and disadvantages of teaching first modelling and verification, and then programming security protocol techniques. In our context, so far, we preferred a bottom-up approach. Since programming was part of the academic background of the students, they could build a viable mental model based on their previous knowledge and experience. However, in institutions where students are already familiar with a more theoretical and rigorous approach to computer science concepts, considering abstract modelling first and then programming may be a viable option.

Finally, although the methodology presented here was mostly discussed in the context of higher education, we think it has the potential to be beneficial also for professional training, in particular programmers without prior specific background in security. Therefore, an investigation in that direction would be appropriate.

Acknowledgements

The author wishes to thank Matt Bishop and Laurence Eagle for their constructive feedback. Part of this work was done while the author was working at the University of Sunderland.

References

Abadi, M. & Fournet, C. (2001). Mobile values, new names, and secure communication (C. Hankin & D. Schmidt, Eds.). In: C. Hankin & D. Schmidt (Eds.), *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17–19, 2001*, ACM. `https://doi.org/10.1145/360204.360213`

Abadi, M., Fournet, C. & Gonthier, G. (2002). Secure implementation of channel abstractions. *Information and computation (Print)*, 174(1), 37–83.

Abadi, M. & Gordon, A. D. (1997). A calculus for cryptographic protocols: The Spi calculus, In: *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1–4, 1997*, ACM. `https://doi.org/10.1145/266420.266432`

ACM and IEEE Computer Society Joint Task Force on Computing Curricula. (2013). *Computer science curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA, Association for Computing Machinery; IEEE Computer Society.

Adams, J. C. (1996). Object-centered design: A five-phase introduction to objectoriented programming in CS1–2, In: *ACM SIGCSE Bulletin*. ACM.

Alexandron, G., Armoni, M., Gordon, M. & Harel, D. (2012). The effect of previous programming experience on the learning of scenario-based programming, In: *Proceedings of the 12th KOLI Calling International Conference on Computing Education Research*, Koli, Finland, ACM. `https://doi.org/10.1145/2401796.2401821`

Alexandron, G., Armoni, M., Gordon, M. & Harel, D. (2014). Scenario-based programming: Reducing the cognitive load, fostering abstract thinking, In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM.

Allodi, L., Cremonini, M., Massacci, F. & Shim, W. (2018). The effect of security education and expertise on security assessments: The case of software vulnerabilities, In: *Workshop on Economics of Information Security*.

Almousa, O., Mödersheim, S. & Viganò, L. (2015). Alice and Bob: Reconciling formal models and implementation. In: C. Bodei, G.-L. Ferrari & C. Priami (Eds.), *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of his 65th Birthday* (pp. 66–85). Springer International Publishing. `https://doi.org/10.1007/978-3-319-25527-9 7`

Atkinson, C. & Kuhne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5), 36–41. `https://doi.org/10.1109/MS.2003.1231149`

Avalle, M., Pironti, A., Pozza, D. & Sisto, R. (2011). JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2(4), 34–48.

Avalle, M., Pironti, A. & Sisto, R. (2014). Formal verification of security protocol implementations: A survey. *Formal Aspects of Computing*, 26(1), 99–123.

AVISPA. (2003). *Deliverable 2.3: The Intermediate Format*. Available at www.avispa-project.org.

Basin, D., Keller, M., Radomirovic, S. & Sasse, R. (2015). Alice and Bob meet equational theories. In: N. Martí-Oliet, P. C. Ölveczky & C. Talcott (Eds.), *Logic, Rewriting, and Concurrency* (pp. 160–180). Springer International Publishing. https://doi.org/10.1007/978-3-319-23165-5 7

Bell, S. (2010). Project-based learning for the 21st century: Skills for the future. *The Clearing House*, 83(2), 39–43.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1), 45–74.

Ben-Ari, M. & Yeshno, T. (2006). Conceptual models of software artifacts. *Interacting with Computers*, 18(6), 1336–1350.

Bhargavan, K., Fournet, C., Gordon, A. D. & Tse, S. (2008). Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 5.

Bishop, M. (2000). Education in information security. *IEEE Concurrency*, 8(4), 4–8.

Bishop, M. (2002). Computer security education: Training, scholarship, and research. *Computer*, 35(4), 31–32.

Bishop, M., Dark, M., Futcher, L., Van Niekerk, J., Ngambeki, I., Bose, S. & Zhu, M. (2019). Learning principles and the secure programming clinic, In: *IFIP World Conference on Information Security Education*. Springer.

Bishop, M. & Frincke, D. A. (2005). Teaching secure programming. *IEEE Security & Privacy*, 3(5), 54–56.

Bishop, M. & Orvis, B. (2006). A clinic to teach good programming practices, In: *Proceedings of the 10th Colloquium for Information Systems Security Education*.

Blanchet, B. (2001). An efficient cryptographic protocol verifier based on Prolog rules, In: *Computer Security Foundations Workshop, IEEE*. IEEE Computer Society.

Blanchet, B. (2012). Security protocol verification: Symbolic and computational models, In: *Proceedings of the First International Conference on Principles of Security and Trust*. Springer-Verlag.

Blanchet, B., Smyth, B. & Cheval, V. (2019). ProVerif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial.

Bloom, B. S. & Krathwohl, D. (1956). Handbook i: Cognitive domain. *New York: David McKay*.

Bruner, J. S. (2009). *The Process of Education*. Harvard University Press.

Bugliesi,M., Calzavara, S.,Mödersheim, S. &Modesti, P. (2016). Security protocol specification and verification with AnBx. *Journal of Information Security and Applications*, *30*, 46–63. https://doi.org/10.1016/j.jisa.2016.05.004

Bugliesi, M. & Focardi, R. (2008). Language based secure communication, In: *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*.

Bugliesi, M. & Modesti, P. (2010). AnBx-Security protocols design and verification, In: *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security: Joint Workshop, ARSPA-WITS 2010*. Springer-Verlag.

Center for Strategic and International Studies. (2016). Hacking the skills shortage. A study of the international shortage in cybersecurity skills [Online; accessed 04 December 2019].

Chomsky, N. (1959). A review of BF Skinner's verbal behavior. *Language*, 35(1), 26–58.

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. *Computer Science Education Research*, 85–100.

Conklin, W. A. & Dietrich, G. (2007). Secure software engineering: A new paradigm, In: *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. https://doi.org/10.1109/HICSS.2007.477

Dark, M., Belcher, S., Bishop, M. & Ngambeki, I. (2015). Practice, practice, practice... secure programmer!, In: *Proceeding of the 19th Colloquium for Information System Security Education*.

Dark, M., Bishop, M., Linger, R. C. & Goldrich, L. (2015). Realism in teaching cybersecurity research: The agile research process (M. Bishop, N. G. Miloslavskaya & M. Theocharidou, Eds.). In: M. Bishop, N. G. Miloslavskaya & M. Theocharidou (Eds.), *Information Security Education Across the Curriculum – 9th IFIP WG 11.8 World Conference, WISE9, Hamburg, Germany, May 26–28, 2015, Proceedings*, Springer. https://doi.org/10.1007/978-3-319-18500-2 1

Dolev, D. & Yao, A. (1983). On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29).

Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M. *et al.*, (2014). The matter of heartbleed, In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM.

Eckerdal, A., McCartney, R., Mostr□om, J. E., Ratcliffe, M., Sanders, K. & Zander, C. (2006). Putting threshold concepts into context in computer science education, In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, Bologna, Italy, ACM. `https://doi.org/10.1145/1140124.1140154`

Frost & Sullivan. (2017). 2017 Global information security workforce study. Benchmarking workforce capacity and response to cyber risk [Online; accessed 04 December 2019].

Garcia, R. & Modesti, P. (2017). An IDE for the design, verification and implementation of security protocols, In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2017, Toulouse, France, October 23–26, 2017*. `https://doi.org/10.1109/ISSREW.2017.69`

Griffiths, R. (2004). Knowledge production and the research-teaching nexus: The case of the built environment disciplines. *Studies in Higher Education*, 29(6), 709–726.

Haberman, B. (2004). High-school students' attitudes regarding procedural abstraction. *EAIT*, 9(2), 131–145. `https://doi.org/10.1023/B%3AEAIT.0000027926.99053.6f`

Haberman, B. & Kolikant, Y. B.-D. (2001). Activating "black boxes" instead of opening "zipper"- a method of teaching novices basic CS concepts, In: *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, Canterbury, United Kingdom, ACM. `https://doi.org/10.1145/377435.377464`

Hadjerrouit, S. (1998). A constructivist framework for integrating the java paradigm into the undergraduate curriculum, In: *ACM SIGCSE Bulletin*. ACM.

Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming, In: *ACM SIGCSE Bulletin*. ACM.

Hadjerrouit, S. (2005). Constructivism as guiding philosophy for software engineering education. *ACM SIGCSE Bulletin*, 37(4), 45–49.

Herzog, J. (2005). A computational interpretation of dolev-yao adversaries. *Theor. Comput. Sci.*, *340* (1), 57–81. `https://doi.org/10.1016/j.tcs.2005.03.003`

(ISC)2. (2019). Global information security workforce study 2019. Cybersecurity professionals focus on developing new skills as workforce gap widens [Online; accessed 04 December 2019].

ISO/IEC. (2010). ISO/IEC 9798-1:2010. Information technology – Security techniques – Entity authentication – Part 1: General.

Johnstone, M. N. (2013). Embedding secure programming in the curriculum: Some lessons learned. *International Journal of Engineering and Technology*, 5(2), 287.

Joint Task Force on Cybersecurity Education. (2017). *Cybersecurity Curricula 2017: Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*. New York, NY, USA, ACM, IEEE, AIS, IFIP.

Jøsang, A., Ødegaard, M. & Oftedal, E. (2015). Cybersecurity through secure software development, In: *IFIP World Conference on Information Security Education*. Springer.

Knobelsdorf, M. (2015). The theory behind theory – computer science education research through the lenses of situated learning. *Informatics in Schools. Curricula, Competences, and Competitions*. `https://doi.org/10.1007/978-3-319-25396-12`

Koppelman, H. & van Dijk, B. (2010). Teaching abstraction in introductory courses, In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, Bilkent, Ankara, Turkey, ACM. `https://doi.org/10.1145/1822090.1822140`

Laschi, R. & Riccioni, A. (2008). Design and implementation of a virtual lab for supporting students in modeling, evaluating and programming secure systems, In: *Proc. of the 13th International Conference on Interactive Computer-Aided Learning (ICL), Villach, Kassel University Press*.

Lebow, D. (1993). Constructivist values for instructional systems design: Five principles toward a new mindset. *Educational Technology Research and Development*, 41(3), 4–16.

Lowe, G. (1997). A hierarchy of authentication specifications. In: *CSFW'97* (pp. 31–43). IEEE Computer Society Press.

Mason, R., Cooper, G. & Wilks, B. (2015). Using cognitive load theory to select an environment for teaching mobile apps development.

McConnell, J. J. (1996). Active learning and its use in computer science. *ACM SIGCSE Bulletin*, 28(SI), 52–54.

McDermott, L. C. & Shaffer, P. S. (1992). Research as a guide for curriculum development: An example from introductory electricity. part i: Investigation of student understanding. *American Journal of Physics*, 60(11), 994–1003.

McGettrick, A. (2013). Toward effective cybersecurity education. *IEEE Security Privacy*, 11(6), 66–68. `https://doi.org/10.1109/MSP.2013.155`

Mirkovic, J., Dark, M., Du, W., Vigna, G. & Denning, T. (2015). Evaluating cybersecurity education interventions: Three case studies. *IEEE Security & Privacy*, 13(3), 63–69. `https://doi.org/10.1109/MSP.2015.57`

Mitre. (2020). CVE (Common Vulnerabilities and Exposures) repository [Online; accessed 04 March 2020].

Mödersheim, S. (2009). Algebraic properties in Alice and Bob notation, In: *International Conference on Availability, Reliability and Security (ARES 2009)*. `https://doi.org/10.1109/ARES.2009.95`

Mödersheim, S. & Viganò, L. (2009). The open-source fixed-point model checker for symbolic analysis of security protocols, Springer.

Modesti, P. (2014). Efficient Java code generation of security protocols specified in AnB/AnBx, In: *Security and Trust Management – 10th International Workshop, STM 2014, Proceedings*.

Modesti, P. (2015). AnBx: Automatic generation and verification of security protocols implementations, In: *8th International Symposium on Foundations & Practice of Security*, Springer.

Moons, J. & Backer, C. D. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education*, 60(1), 368–384. `https://doi.org/10.1016/j.compedu.2012.08.009`

Pothamsetty, V. (2005). Where security education is lacking, In: *Proceedings of the 2nd Annual Conference on Information Security Curriculum Development*, Kennesaw, Georgia, ACM. `https://doi.org/10.1145/1107622.1107635`

Pullen, J. M. (2001). The network workbench and constructivism: Learning protocols by programming. *Computer Science Education*, *11* (3), 189–202.

Quevedo-Torrero, J. U. (2009). Learning theories in computer science education, In: *Information technology: New Generations, 2009. ITNG'09. Sixth International Conference on*. IEEE.

Schmidt, B., Meier, S., Cremers, C. & Basin, D. (2012). Automated analysis of Diffie-Hellman protocols and advanced security properties, In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE.

Schneider, F. B. (2013). Cybersecurity education in universities. *IEEE Security & Privacy*, 11(4), 3–4.

Shaffer, D., Doube, W. & Tuovinen, J. (2003). Applying cognitive load theory to computer science education, In: *Workshop of the Psychology of Programming Interest Group*.

Sivilotti, P. A. & Lang, M. (2010). Interfaces first (and foremost) with Java, In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, Milwaukee, Wisconsin, USA, ACM. `https://doi.org/10.1145/1734263.1734436`

Skinner, B. F. (1954). The science of learning and the art of teaching. *Cambridge, Mass, USA*, 99–113.

Sweller, J. (2016). Cognitive load theory and computer science education, In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM.

Tabassum, M., Watson, S. & Richter Lipford, H. (2017). Comparing educational approaches to secure programming: Tool vs. TA, In: *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, Santa Clara, CA, USENIX Association.

Theisen, C., Williams, L., Oliver, K. & Murphy-Hill, E. (2016). Software security education at scale, In: *Proc. IEEE/ACM 38th Int. Conf. Software Engineering Companion (ICSE-C)*.

Tsipenyuk, K., Chess, B. & McGraw, G. (2005). Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security Privacy*, 3(6), 81–84. `https://doi.org/10.1109/MSP.2005.159`

Wadsworth, B. J. (1996). *Piaget's Theory of Cognitive and Affective Development: Foundations of Constructivism*. Longman Publishing.

Walden, J. & Frank, C. E. (2006). Secure software engineering teaching modules, In: *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, Kennesaw, Georgia, ACM. `https://doi.org/10.1145/1231047.1231052`

Walker, G. N. (2004). Experimentation in the computer programming lab, In: *Working Group Reports from ITICSE on Innovation and Technology in Computer Science Education*, Leeds, United Kingdom, ACM. `https://doi.org/10.1145/1044550.1041660`

Wen, S.-F. & Katt, B. (2019). Toward a context-based approach for software security learning. *Journal of Applied Security Research*, 1–20.

Williams, K. A., Yuan, X., Yu, H. & Bryant, K. (2014). Teaching secure coding for beginning programmers. *Journal of Computing Sciences in Colleges*, 29(5), 91–99.

Yurcik, W. & Doss, D. (2001). Different approaches in the teaching of information systems security. In: *Proceedings of the Information Systems Education Conference*.

P. Modesti is a senior lecturer in Cybersecurity at the School of Computing, Engineering and Digital Technologies at Teesside University, United Kingdom. He holds a PhD in Computer Science from Ca'Foscari University Venice, Italy, and is a fellow of Advance Higher Education. His main research interests include applied formal methods for security, languages and tools for security.