Formal Methods for Security Knowledge Area
Version 1.0.0

David Basin | ETH Zurich

EDITOR
Steve Schneider | University of Surrey

REVIEWERS
Rod Chapman | Protean Code Ltd
Stephen Chong | Harvard School of Engineering and Applied Sciences
Mark Ryan | University of Birmingham
Andrei Sabelfeld | Chalmers University of Technology

COPYRIGHT

The CyBOK project would like to understand how the CyBOK is being used and its uptake. The project would like organisations using, or intending to use, CyBOK for the purposes of education, training, course development, professional development etc. to contact it at **contact@cybok.org** to let the project know how they are using CyBOK.

Version 1.0.0 is a stable public release of the Formal Methods for Security Knowledge Area.

CHANGELOG

Version date	Version number	Changes made
July 2021	1.0	

INTRODUCTION

This Knowledge Area surveys the most relevant topics in formal methods for security. As a discipline, formal methods address foundations, methods and tools, based on mathematics and logic, for rigorously developing and reasoning about computer systems, whether they be software, hardware, or a combination of the two. The application of formal methods to security has emerged over recent decades as a well-established research area focused on the specification and proof of security properties of systems, their components, and protocols. This requires a precise specification of:

- the **system** at an appropriate level of abstraction, such as design or code,
- the **adversarial environment** that the system operates in, and
- the **properties**, including the **security properties**, that the system should satisfy.

Formal reasoning allows us to prove that a system satisfies the specified properties in an adversarial environment or, alternatively, to identify vulnerabilities in the context of a well-defined class of adversaries.

Formal methods have a wide scope of applicability. Hence this Knowledge Area is relevant to many other KAs as it encompasses general approaches to modelling, analysis, and verification that relate to many technical aspects of cybersecurity. Moreover, as formal methods for security have applications across the entire system stack, this KA leverages background knowledge of hardware and software security from other chapters. As specific prerequisites, this KA benefits from background knowledge of logic, discrete mathematics, theorem proving, formal languages, and programming semantics, at a Computer Science undergraduate level, and provides references covering these topics; some relevant text books include [1, 2, 3]. Modelling and abstraction are cornerstones of formal methods. This KA covers their application across security topics, including access control, secure information flow, security protocols, and program correctness. These can be considered with respect to requirements such as authentication, confidentiality, anonymity, and integrity, and in the context of specific attacker models that capture different classes of attacker capabilities, giving rise to threats to the system.

We cover a variety of approaches to formal analysis and verification, including those founded on semantics, games, simulation, equivalence, and refinement. This includes both logic-based approaches (where requirements are expressed as logical statements) and behavioural approaches (given by models of secure behaviour). Our emphasis is on state-of-the-art practical applications of formal methods, enabled by tool support. Hence, the KA covers representative examples of mature tools for these approaches as applied in practice, including general-purpose theorem provers such as Isabelle/HOL and Coq, satisfiability solvers such as Z3, model checkers such as SPIN, FDR, and PRISM, and more specialised security-specific verification tools such as Tamarin, ProVerif, CryptoVerif, and EasyCrypt. We also cover representative examples of tool support for program development and code analysis.

Finally, we provide real-world examples where formal methods have been effective in verifying the security of systems and their components or where analysis has been instrumental in identifying vulnerabilities.

Structure. We have structured this KA along three dimensions: foundations and methods for modelling systems, types of systems, and level of abstraction. After motivating the use of formal methods for security, we survey the different foundations and methods, along with associated tools. We subsequently consider their application to different kinds of systems, as well as differentiating between levels of abstraction, such as programs versus designs. In particular, we explore hardware, protocols, software and large-scale systems, and system configuration. These categories overlap, of course (e.g., systems are built from hardware and software) and a formal method may be used across multiple categories (e.g., methods for the analysis of side-channels or secure information flow are used for both hardware and software). Nevertheless, these categories serve as a convenient structure to introduce the different approaches, and to highlight their scope and reach with respect to different formal analysis problems.

CONTENT

1 MOTIVATION

Formal methods have come a long way since the pioneering work of Floyd, Dijkstra, and Hoare on assigning meaning to programs and using deductive methods for the formal verification of small imperative programs.

Despite the undecidability of the underlying verification problems, formal methods are being used with increasing success to improve the security of large-scale, real-world systems. Examples of the successful industrial usage of formal methods are found at companies such as Microsoft [4], Amazon [5, 6], and Google [7]. Concrete examples are given later in this KA.

There are several motivations for the use of formal methods in security, as discussed next.

1.1 Inadequacy of Traditional Development Methods

System development often follows the traditional cycle of code, test and fix. For security-critical systems, "test and fix" becomes "penetrate and patch" as errors can lead to security vulnerabilities. For example, a program missing a check when writing into memory can be exploited by a buffer-overflow attack. Alternatively, a design that fails to check all access requests can lead to unauthorised resource usage. The effects of even small errors can be disastrous in practice. As explained in the Secure Software Lifecycle CyBOK Knowledge Area [8], some attackers are very skilled at finding and exploiting even obscure bugs. Formal methods offer the possibility of improving this cycle by rigorously establishing that systems meet their specification or are free of certain kinds of bugs.

As we shall see, formal methods can be used during different phases of system development and operation, and can be applied to different kinds of system artifacts.

- **System design**: They can be used to specify and verify (or find errors in) designs of all kinds. A good example is security protocols, considered in Section 4, where formal methods have made great strides in improving their security.
- **Code level**: As explained in the Software Security CyBOK Knowledge Area [9], programs have bugs, which often represent security vulnerabilities. Indeed, with the right tools, finding bugs is easy [10, 11]. As we describe in Section 2.3, formal methods tools for

code range from simple static analysers that verify "shallow" properties (e.g., ensuring the absence of specific kinds of security bugs) in very large code bases, to full-fledged interactive verification that can verify deep properties, but usually in smaller programs. We consider the application of formal methods to programs of different kinds in Section 5.

- **Configuration level**: The security of systems also depends on how they are configured. For example, access control mechanisms require a specification of who is authorised to do what. We address the application of formal methods to configurations in Section 6.

1.2 Towards More Scientific Development Methods

The previous discussion positions formal methods as a way to reduce security-relevant bugs. However, one can approach the problem from the other end: as a quest to place the development of secure systems on a firm mathematical ground. This quest for mathematical methods to develop programs (more generally, systems) that behave as they should, rather than merely to catch bugs, was a driving motivation for the early pioneers in formal methods. The essence of this position is that programs and systems can be viewed as *mathematical objects* that can be specified and reasoned about using mathematics and logic. For example, imperative programs can be given semantics — operational, denotational, or axiomatic [12] — which can be embedded in a theorem prover and used to formalise and reason about programs and their properties [3]. Under this view, specifications are essential for documenting what programs should do and proofs are essential for ensuring that the programs do it.

Both motivations for formal methods, in this subsection and the previous one, are standard and independent of security. But security adds several challenges. First and foremost, the environment is adversarial: one assumes it contains *adversaries* (also known as *attackers*) who try to attack the system and possibly violate its intended security properties. See the Adversarial Behaviours CyBOK Knowledge Area [13] for a discussion on the kinds of adversaries and adversarial behaviours one finds in practice. Indeed, security is only meaningfully understood with respect to an *adversary model* that specifies a class of attackers in terms of their capabilities or behaviours. To underscore this point, consider protecting information on a computer's disk. Network and file system access control might protect the information on the disk against a remote network attacker, but fail to protect it against an attacker with physical access to the computer who can remove the disk, read out its contents, and even forensically analyse it to reconstruct deleted files. Hence when reasoning about systems and their security, part of the specification must include a model of the adversary; either this is made explicit (for example, when reasoning about security protocol in Section 4), or it is implicit and part of the argument used to justify the analysis method.

Second, some security properties differ from those properties considered in traditional program correctness in that they are not trace properties. A *trace property* is a property (or set) of individual system executions. In contrast, some security properties are *hyperproperties* [14] defined as properties of sets of executions. Note that interest in hyperproperties is not unique to security; hyperproperties have been studied in other contexts such as process calculi. However, hyperproperties play a particularly prominent role in security in specifying central notions like confidentiality or integrity. We will return to the distinction between properties and hyperproperties in Section 2.

In the quest to raise the level of rigour for security arguments, some researchers have called for the development of a *science of security*. Such a science could embrace not only rich specifications and verification methods, but also laws for reasoning about security that relate

systems and adversaries with security properties (or policies). Taking an example from [15]:

> "For generality, we should prefer laws that relate classes of attacks, classes of defenses, and classes of policies, where the classification exposes essential characteristics. Then we can look forward to having laws like "Defenses in class $\mathcal{D}$ enforce policies in class $\mathcal{P}$ despite attacks from class A" or "By composing defenses from class $\mathcal{D}'$ and class $\mathcal{D}''$, a defense is constructed that resists the same attacks as defenses from class $\mathcal{D}$."

The question is still open on how such a science should look and how it would relate to traditional sciences, with their notions of theories about the empirical world and refutation via experimentation and observation. Indeed, an in-depth analysis and critique of different approaches to developing a science of security was provided by [16] who note, for example, that the history lessons from other sciences first need to be better accounted for. In addition, formal verification, while unquestionably valuable, is only one part of a security rationale. One often fails to validate the mapping between models and assumptions to actual systems and adversaries in the real world [17]; in fact, this validation between mathematical abstractions and the real world falls outside of the domain of formal proof [18]. A real-world adversary will not be deterred from attacking a system just because a mathematical theorem apparently rules this out.

1.3 Limitations

Section 1.2 touches upon a central limitation of formal methods: the faithfulness of the models. Namely, one verifies properties of models, not the actual systems used. Hence the quality of the results depends on the quality of the model. This is a general problem for all formal methods, although runtime verification methods, discussed in Section 2.3.4, partially mitigate this problem as they analyse the actual running system rather than a separate model of the system. The use of executable system models, supporting simulation and model animation, can also help uncover system formalisation errors. For security, adversary modelling is particularly critical and the capabilities of adversaries are notoriously difficult to model accurately given human skill and creativity in attacking systems.

A second limitation of formal methods is that one may simply fail to specify all relevant system requirements, including relevant security requirements. Requirements may also be specified incorrectly. Formal methods that return counter examples, such as model checkers, can to some extent help counter the problem of incorrect property specifications (or mistakes in the system model), but they will not help in revealing overlooked system properties. Bridging requirements, which are in the heads of the different system stakeholders, with formal models is a difficult problem that is ultimately outside of the domain of formal methods themselves.

To illustrate the above two points, consider full-stack verification, where the idea is to verify as much of the entire system as possible, rather than just fragments of it. In particular, the aim is to bridge layers of the software-hardware stack such as high-level programs, low-level assembly programs, the operating system that programs run on, and the underlying hardware. In Section 5.6 we give an example of such a system stack where all levels are linked by theorems about the translations provided by compilers and assemblers. The theorems and their proofs provide rigourous mathematical guarantees about the stack's contents (e.g., programs, assembly code, operating system, and hardware) and the relationships between the stack's layers. But the two limitations mentioned above, concerning the top and bottom of the stack, remain. That is, are the relevant properties correctly specified for the high-level

programs and does the low-level hardware actually conform to its model?

Finally, there are theoretical and practical limitations regarding the verification methods and tools themselves. Rice's theorem tells us that any nontrivial property about Turing complete languages is undecidable. So we should not hope for a push-button verification tool that will always terminate and can verify all relevant properties of arbitrary systems in a sound and complete way. In practice, compromises must be made. The tools may approximate behaviours (giving false negatives or false positives depending on the approximation, while still being useful), or target only specific classes of programs or properties, or require human interaction, as is the case of interactive theorem proving. Another drawback is that many tools are demanding in terms of the expertise and effort required, and these demands may be hard to satisfy in projects with limited resources for quality assurance.

2 FOUNDATIONS, METHODS, AND TOOLS

[14, 19, 20, 21, 22, 23]

In this section we will take a broad, high-level view of approaches to reason about the security of systems. Note that by *system*, we include hardware and software at all levels of the system stack, and at different levels of abstraction. In some cases, we will move between the terms system, program, and process, to reflect the terminology used in the relevant literature.

2.1 Properties of Systems and Their Executions

2.1.1 Trace Properties

In formal methods, it is common to take an abstract view of systems and their behaviours. System executions are modelled as finite or (for non-terminating systems) infinite sequences

$$\pi \triangleq s_0\, s_1\, s_2\, \cdots ,$$

where the s_i belong to an alphabet Σ. Possible interpretations of the s_i are system states, atomic actions, or even state-action pairs. Such a sequence is called a *trace*.

Under this view, a system defines a set of traces. A *property* can also be defined by a set of traces, and is, unsurprisingly, called a *trace property*. System correctness can then be defined in terms of set inclusion: the traces of the system are included in the traces of the property, i.e., the system's behaviour satisfies the property.

In security, many relevant properties are *safety properties*. Informally they stipulate that something "bad" or "undesirable" does not happen during system execution [24, 19]. An invariant is a standard example of a safety property, expressing that the system never reaches an undesired state. In more detail, an invariant is a property of states that should hold of every reachable system state. It is defined by a subset of Σ (the "good" states) and can be extended to traces $s_0\, s_1\, s_2\, \cdots$, whereby a trace satisfies an invariant when every s_i satisfies it. An example is the invariant "only the owner of a file may view its content"; this is a safety property where the bad thing is reaching a system state where a file is being viewed by someone other than its owner. Another example of a safety property, expressed directly as a trace property and formalisable in a temporal logic like linear temporal logic (LTL), would be that "withdrawing funds requires first entering a PIN." In temporal logic this might be expressed as:

$$\Box(\text{FundsWithdraw} \rightarrow \blacklozenge\text{EnterPIN}) ,$$

where the two propositions are interpreted as the actions of withdrawing funds and PIN entry and $\Box$ and $\blacklozenge$ are the LTL operators for "always" and "sometime in the past", respectively. Note that this safety property is not an invariant as it is not a property of states.

The notion that something bad never happens seems to be a good fit for many security properties, where the bad thing might be some unauthorised action. Safety properties also have the attractive feature that, when they are violated, they are finitely falsifiable. Namely, a finite trace is enough to witness a violation since once a safety property is violated, the breach can never be rectified by some extension of the trace.

Not all trace properties are safety properties. In particular, *liveness properties* do not constrain the finite behaviour of a system but rather impose requirements on its infinite behaviours. A typical example is that some event occurs infinitely often or that, possibly under some conditions, something "good" eventually happens. In practice, liveness properties are much less common in security than safety properties as they abstract from *when* events occur. Although this is meaningful in verification, e.g., to rule out infinite loops, in security we often need more concrete guarantees. For example, if an authentication server should respond to requests, it is not that helpful to know that it will *eventually* do so. In practice some upper bound on the response time is required, which thereby turns the property into a safety property. Finally, note that some trace properties are neither safety nor liveness, but rather the conjunction of a safety and liveness property [19].

Trace properties can be established using model checkers, notably when the systems generating them are finite-state, or using theorem provers. For safety properties, their finite falsifiability also makes them well-suited to runtime monitoring. We describe this in more detail in Section 2.3.

2.1.2 Hyperproperties

Some security properties are not properties of traces but rather hyperproperties, which are properties of *sets* of traces. To determine whether a system satisfies a hyperproperty it is no longer sufficient to examine each of the system's traces individually, one must instead examine the entire set of traces.

Let us start with a simple illustrating example, from the domain of side-channel analysis, also discussed in Section 3.2. Consider timing side-channels in a setting where adversaries cannot only observe system input and output, but also how much time functions, like encryption, take to be executed on their input. This could be modelled using *timed traces*: events modelling function computations are augmented with the time required for the computation. If a function's computation does not admit a timing side-channel, then the time required to compute the function should be independent of any secret input. In other words, the time taken to execute on any secret is the same as the time taken to execute on any other secret. Analysing any individual trace π would be insufficient to establish this. One must examine the set of all of the system's traces. Actually, in this particular example, it would suffice to examine all *pairs* of system traces. This is an example of what is known as a *2-safety hyperproperty* [14].

Hyperproperties were first studied, without being so named, in the 1980s when researchers started investigating noninterference [20] and related secure information flow properties [25, 26]. Hyperproperties emerged almost 30 years later [14] as a general formalism capable of specifying such notions. As such, hyperproperties represent a generally useful contribution from the security community to the formal methods community. Let us consider this in more

detail focusing on noninterference, the prevailing semantic notion for secure information flow [27]; see also Section 5.1 for more on this topic.

Noninterference is studied in a setting where users and the data they input into systems have different classifications and the actions of privileged users should not affect, or influence, those of less privileged users. In other words, the secret (or high) inputs of privileged users should not interfere with the public (low) outputs observed by non-privileged users. Put more simply, the public outputs are independent of the secret inputs.

To see how this can be formulated as a hyperproperty, consider a system with inputs I that receives high inputs h and low inputs $i \in I \backslash \{h\}$ and produces low outputs o. In this context, we might formalise noninterference by requiring that all system traces π and π' whose inputs differ only in h, have, at all times, the same outputs o. This is a hyperproperty whose formalization involves two traces (technically, it is another example of a 2-safety hyperproperty) and it can be formalised in a hyper-temporal logic supporting explicit quantification over traces, like hyperLTL [28, 29] as follows:[1]

$$\forall \pi . \forall \pi' . \Box (\bigwedge_{i \in I \backslash \{h\}} i_\pi = i_{\pi'}) \Rightarrow \Box (o_\pi = o_{\pi'}) \, .$$

The original formalisation of noninterference by Goguen and Meseguer [20] was in the spirit of the above definition, although somewhat different in its details. It was stated in terms of deterministic state-transition systems and the purging of high inputs. Namely, the commands entered by users with high clearances can be removed without affecting the outputs learned by those users with low clearances. Since this original formulation, researchers have proposed numerous generalisations and variants [27], which can naturally be cast as hyperproperties. These include observational determinism [30] (stating that every pair of traces π and π' with the same initial low observation are indistinguishable for low users, i.e., the system appears deterministic to these users), generalised noninterference [31] (which allows for nondeterminism in the low outputs, but requires that they are unaffected by high inputs) and numerous variants of noninterference based on interleavings of traces, such as those considered by McLean [32] and other researchers.

Hyperproperties have their own associated theory with notions like hypersafety, which generalises safety properties to sets of traces, and hyperliveness. For example, a hyperproperty is hypersafety when, to disprove it, it suffices to show a counterexample set of finite traces. There are also model checkers explicitly built for hyper-temporal logics, discussed further in Section 2.3.

[1] In this hyperLTL formulation, taken from [29], the π and π' are path variables (semantically, they range over the infinite traces of a given Kripke structure) and atomic formulas of the form a_π refer to the occurrence of an atomic proposition a at the current (starting) position of the path π.

2.1.3 Relations on Systems

The previous notions focused on properties expressed in terms of system behaviours, i.e., their execution traces. One may alternatively define relations directly between systems themselves. A standard setting for this is where a system, possibly concurrent or distributed, is represented as a *labelled transition system* (LTS) or some variant thereof. In its simplest form, an LTS is a triple $\langle Q, A, \rightarrow \rangle$ consisting of a set of states Q, a set of actions (or events) A, and a transition relation $\rightarrow \subseteq Q \times A \times Q$, where $q \xrightarrow{a} q'$ represents the transition from state q to q' taking place when the action a occurs. An LTS may be extended to a *labelled Kripke structure* by additionally labelling states by predicates, from a given set P, that hold in that state. When P is a singleton (or omitted) then the labelled Kripke structure is simply called a *Kripke structure*. Moreover, when the labelled Kripke structure has an initial state, it is sometimes called a *process graph*. Finally, it is called a *(nondeterministic)* finite automaton when the states and set of actions are finite and the single predicate denotes acceptance.

There has been substantial research in the formal methods community on *process theory*: how to represent systems by processes and define and verify statements about the relationship between processes. In addition to directly representing processes as variants of labelled transition systems, one can use richer or more specialised languages that emphasise aspects like concurrency or communication. Options here include structures such as Petri nets, event structures, or process calculi like CCS and CSP.

Numerous relations have been defined and studied to compare processes. They capture different notions like two processes are interchangeable or one process implements another, for instance. Such relations range from being very strong, like the isomorphism of process graphs, to being much weaker, like the equivalence of their traces, viewing them as automata, as well as relations in between like simulation and bisimulation [33, 34, 35]. Demanding that two systems have the same structure (isomorphism) is usually too strong in practice as this is not something that a system's user, or an attacker, can directly observe. The weaker properties of trace equivalence or trace containment are relevant for checking safety properties, where one LTS represents an implementation and the other a specification. Bisimulation is stronger than trace equivalence as it additionally identifies systems with the same branching structure.

Formalising properties as process equivalences is common in security. Different equivalences are used, such as observational equivalence (discussed below), testing equivalence, and trace equivalence [36, 37, 38]. A standard cryptographic notion is *indistinguishability*, which formalises that an adversary cannot distinguish between two protocols, usually either the same protocol using different secrets, or a real protocol and a simulation of the protocol using random data. Observational equivalence formalises a similar notion in the context of processes, whereby two processes P and P' are observationally equivalent if, intuitively speaking, an observer, who may be an attacker, cannot tell them apart. The processes may compute with different data and the way they compute internally may be completely different, but they appear identical to an external observer. Observational equivalence is also attractive as it is a congruence relation that supports compositional proofs: whenever a process P is equivalent to another process P', then P can be replaced by P' in other, more complex processes.

Observational equivalence for processes in the applied pi calculus [38] has been extensively studied in the formal methods for security community. This calculus is a process calculus, based on Milner's pi calculus, that supports a term algebra used to model cryptographic operations used, for example, in cryptographic protocols. It is worth noting that robust verification tools exist for this calculus like the security protocol model checker ProVerif [39],

discussed in Section 4. For the applied pi calculus, observational equivalence amounts to a form of labelled bisimulation, which provides proof techniques for establishing process equivalence. Moreover, observational equivalence can be naturally used to formalise privacy-style properties. Applications include formalising the resistance of protocols to guessing attacks, strong secrecy (the attacker cannot distinguish between different values of a secret), and anonymity and unlinkability properties (for example, the attacker cannot distinguish between two processes implementing an election protocol that are otherwise identical, except that one swaps the votes of two arbitrary voters) [40, 41, 39].

2.2 Logics and Specification Languages

Formal methods require one or more languages to specify systems and their properties. In practice, one rarely specifies systems directly as labelled transition systems as it is easier to use higher-level languages. Even programming languages or hardware description languages can be used, where programs are given operational semantics in terms of transition systems and there is support for translating between high-level and low-level descriptions. Alternatively, one might work with a specialised language like a process calculus that emphasises particular aspects of systems, such as concurrency and interaction. We have already mentioned examples of specialised languages, such as the applied pi calculus, which is well suited for specifying security protocols.

In many cases, the specification language is the language of a logic (or a logical theory) that also provides a sound way to reason about statements in the language, in particular, to determine their validity or satisfiability. The specific language used and statements made depend on what is being formalised and what should be proven.

There is no general agreement on what is the ideal logic or language to use for program specification; one's choice of logic and associated verification system is partly a matter of taste, expressiveness, desire for automation, and education. The specification formalisms used in practice range from weak, relatively inexpressive logics that have decision procedures, to stronger, expressive logics that usually require interactive theorem proving. Weaker logics include propositional logic and propositional temporal logics. Stronger expressive logics include higher-order logics, both classical and constructive. An intermediate option would be to use a logic based on (some fragment of) first-order logic with equality and background theories.

The weaker logics are limited in what they can formalise. For example, propositional logic cannot talk about relations and functions. This is possible in first-order logic, but only for some relations and functions. For instance, one cannot formulate inductively defined relations, which are relevant when reasoning about the operational semantics of systems or protocols, e.g., to capture the reachable states. When the logic used is insufficiently expressive, then such notions must be approximated. For example, using the technique of bounded model checking, one can encode in propositional logic the traces of a system up to some given length and then use decision procedures based on satisfiability or SMT solvers to verify properties of these bounded-length traces [42].

A middle ground is to use a specification language based on a particular fragment of first-order logic or a given first-order theory such as primitive recursive arithmetic or Peano arithmetic, which would allow formalising recursive computations over the natural numbers and other data structures. The ACL2 theorem prover [43] is an example of a theorem prover built on such a logic that has been used to formalise algorithms and systems represented in pure Lisp.

The logic used enables a high degree of automation in constructing proofs, in particular for proofs by mathematical induction. ACL2 has been applied to verify the functional correctness and security properties of numerous industry-scale systems [44].

Alternatively, one might prefer a richer logic, trading off proof automation for expressiveness. Higher-order logic is a popular example. It can either be based on classical higher-order logic, formalised, for example, in the Isabelle/HOL [45] or HOL-light [46] systems, or a constructive type theory like the calculus of inductive constructions, implemented in the Coq tool [47, 48]. In these richer logics, one can formalise (essentially) all mathematical and logical notions relevant for reasoning about systems. These include systems' operational semantics, properties, hyperproperties, and other correctness criteria, as well as relations between systems like refinement relations. Nipkow and Klein's textbook on concrete semantics [3] gives a good overview of how programming language semantics, programming logics, and type systems can be formalised in Isabelle/HOL, with applications, for example, to information flow control.

2.3 Property Checking

Arguments about systems' security properties can be machine supported. This ranges from tool-supported audits, used to detect potential vulnerabilities, to proofs that systems are secure. We will explore a range of different options in this section, including those based on proofs, static analysis, and dynamic analysis. All of these improve upon human-based system audits or pencil-paper based security proofs in their precision, automation and (to varying degrees) scalability.

2.3.1 Interactive Theorem Proving

Interactive theorem proving is the method of choice for proving theorems when using expressive logics like higher-order logic. In general, interactive theorem proving is used for logics and formalisms where deduction cannot easily be automated, and therefore, humans must interactively (or in a batch mode) guide a theorem prover to construct proofs. This is often the case when inference problems are undecidable or of sufficient computational complexity that human assistance is required to construct proofs in practice.

In contrast with the other methods described in this section, interactive theorem proving is substantially more labor-intensive. Some of the more tedious forms of interaction (arithmetic reasoning, simple kinds of logical reasoning, applying lemmas, etc.) can be offset by integrating decision procedures and other inference procedures into theorem provers. Moreover, many provers are extensible in that users may write tactics, which are programs that implement proof construction strategies and thereby automate parts of proof construction [49]. Nevertheless, even with automation support, it may still be necessary to guide the theorem prover by establishing lemmas leading up to a proof. For example, for program verification, one may need to provide loop invariants, or appropriate generalisations for proving inductive theorems. This effort, although time-consuming, has a side benefit: the human carrying out the proof gains insight into why the theorem holds.

Interactive theorem proving has been applied to numerous large-scale verification projects where the verification time is measured in person-years. We will present one example of this, the verification of the seL4 microkernel, in Section 5.4.

2.3.2 Decision Procedures

Advances in decision procedures and other forms of algorithmic verification have played an essential role in increasing the usage and acceptance of formal methods across a large range of industry applications. Even humble propositional logic is relevant here, as it often suffices to encode and reason about (finite) system behaviours, system configurations, and other system artifacts. There now exist highly effective constraint solvers for the satisfiability problem (SAT) in propositional logic. Due to advances in algorithms, data structures, and heuristics such as conflict-driven backtracking, SAT solvers like Chaff [50], Grasp [51], and MiniSAT [52] can determine the satisfiability of formulas with millions of clauses and solve real problems in program analysis.

SAT-based procedures (as well as other procedures, such as those used for computer algebra [53]) have also been successfully extended to reason about the satisfiability of fragments of first-order logic, so-called satisfiability modulo theories (SMT) [54, 21] as embodied in tools like Z3 [55], CVC4 [56], and Yices [57]. The languages that such tools handle are fragments of first-order logic restricted syntactically or semantically, e.g., by syntactically restricting the function or predicate symbols allowed or by fixing their interpretation. Such restrictions can lead to decidable satisfiability problems and allow for specialised algorithms that exploit properties of the given fragment. The resulting decision procedures may work very well in practice, even for fragments with high worst-case computational complexity. Examples of theories supported by SMT solvers include linear and nonlinear arithmetic (over the reals or integers), arrays, lists, bit-vectors, IEEE standard floating-point arithmetic, and other data structures.

SAT and SMT procedures have numerous applications for reasoning about properties of systems. These include determining the validity of verification conditions, symbolic execution, bounded and unbounded model checking, software model checking, predicate abstraction, and static analysis, to name but a few examples. We shall see examples in other sections of this chapter of how decision procedures are used to support other deduction methods.

Many security properties can be expressed as temporal properties, in particular safety properties. When systems, or their abstractions as models, are finite-state, then model checking algorithms provide a decision procedure for determining that the system model satisfies the specified properties [58, 59]. For small state spaces, one may explicitly represent and exhaustively search them. For larger state spaces, more sophisticated techniques have been developed. For example, both the system and property can be represented as automata and efficient algorithms are used to determine if the system specification conforms to the property, for notions of conformance that correspond to language inclusion, different refinement orderings, or observational equivalence. The transition systems of the automata can be represented symbolically, e.g., using binary decision diagrams [60], and strategies like partial-order reduction [61] can be used to reduce the search through the state-space of concurrently executing automata.

Numerous successful model-checkers have been developed. Aside from incorporating different algorithms and data structures, they employ different system and property specification languages. System specification languages include variants of automata, process algebra, and higher-level programming languages for describing concurrently executing processes. Property specification languages include temporal logics like LTL, CTL, and probabilistic variants. Examples of effective, general purpose model checkers are the explicit-state LTL model checker SPIN [62], the symbolic model checker NuSMV supporting both LTL and CTL [63], the model checker FDR2 supporting programs and properties specified in the process calculus

CSP and different refinement relations [64], and the model checker PRISM for modelling and reasoning about systems with random or probabilistic behaviours [65].

Particularly relevant for security are algorithms and tools that have been developed for model checking systems with respect to hyperproperties [29]. Also relevant are the specialised model checkers developed for security protocols, described in Section 4.1.2. These tools support protocol specifications involving cryptographic functions and effectively handle the non-determinism introduced by an active network adversary.

2.3.3 Static Analysis

Static analysis (cf. Malware & Attack Technologies CyBOK Knowledge Area [66] and Software Security CyBOK Knowledge Area [9]) refers to a broad class of automated methods that analyse programs statically, that is without actually executing them. Most static analysis methods operate on a program's source code or some kind of object code, rather than at the level of designs. Despite the undecidability or intractability of most questions about program behaviour, static analysis attempts to efficiently approximate this behaviour to either compute sound guarantees or to find bugs. The behavioural properties analysed are usually limited to specific problems like the detection of type errors or division-by-zero. For security, the properties might focus on vulnerabilities such as injection attacks or memory corruption problems. In contrast to verification using theorem provers, the emphasis is on completely push-button techniques that scale to industry-sized code bases. Indeed, in the interest of minimal user interaction (and thereby greater acceptance), many static analysis methods do not even require a formal specification but instead target certain kinds of "generic" problems that often lead to runtime exceptions or security vulnerabilities in practice. In other words, they target "shallow", but meaningful, general properties analysed on huge code bases rather than "deep" system-specific properties analysed on relatively small programs and designs.

The methods used vary from tool to tool. As observed in the Software Security CyBOK Knowledge Area [9] a major divide in the tools, and also in the research community, concerns soundness. Some tools are heuristic-based checkers and explicitly reject the requirement for soundness [11]. For other tools, soundness is imperative and, while false positives may be tolerated, false negatives are not acceptable.

Many of the sound methods can be cast as some kind of abstract interpretation [22] where the program's control-flow graph is used to propagate a set of abstract values through the different program locations. The abstract values are approximate representations of sets of concrete values — for example, the abstract values may be given by types, intervals, or formulas – and are determined by an abstraction function, which maps concrete values to abstract values. The propagation of abstract values is continued until these values cease to change. Mathematically, this can be understood as the iterative application of a monotone function until a fixed point is reached.

Industrial-strength static analysis tools are extremely successful and widespread. Early examples of such tools, from the 1970s, were type checkers and programs like LINT, which enforced the typing rules of C even more strictly than C compilers would. More modern tools leverage theoretical advances in abstract interpretation, constraint solving, and algorithms for inter-procedural analysis. Examples of such tools include Grammatech's CodeSonar, Coverity's Prevent (for an early account, see [11]), or Micro Focus' Fortify tool.

As a concrete example of a successful tool, the Fortify tool classifies security bugs into different categories and employs specialised analysis techniques for each category. As

an example, one category concerns input validation, and includes vulnerabilities such as buffer overflows, command injections, cross-site scripting, HTTP response splitting, path manipulation, reflection abuse, and improper XML validation, to name but a few. One static analysis technique that the Fortify tool applies to these problems with considerable success is *taint analysis* [2], which tracks how possibly untrusted inputs can flow unchecked through the program and thereby affect (or taint) arguments to function calls or outputs to users. For instance, if user input could flow through the program and be used as part of an SQL query, then this would represent a potential SQL injection vulnerability. Other categories, with associated analysis techniques, focus on API abuse, improper use of security features, or encapsulation issues.

Two modern examples of the large-scale applications of formal methods are the use of static analysis (and other) tools at Google and Amazon. Google incorporates a variety of analysis tools into their developer workflow [7]. The tools are, by the authors' own account, "not complex" and include bug finding tools that extend the compiler (including abstract-syntax-tree pattern-matching tools, type-based checks, and unused variable analysis) and more sophisticated tools that support code audits. Unlike the compiler-based tools, the audit tools incorporate abstract-interpretation techniques that may generate false positives, that is, they sometimes report on potential problems that are not actual problems. The Google tools also suggest possible fixes to the problems they report.

Amazon [6, 67] has similarly had considerable success applying formal methods to prevent subtle but serious bugs from entering production. In contrast to Google's practices, as reported in [7], Amazon uses a combination of automated static analysis tools alongside techniques based on explicit specification, model checking and theorem proving. So push-button techniques are combined with more time-intensive specification and verification for critical components or properties. For example, multiple Amazon teams are using TLA+ (which is a language based on a combination of set theory and temporal logic, with associated tools) to specify systems and carry out proofs about their properties. [68] reports that these tools are extensively used to reason about security-critical systems and components, including cryptographic protocols, cryptographic libraries, hypervisors, boot-loaders, firmware and network designs.

2.3.4 Dynamic Analysis

Runtime verification (cf. Malware & Attack Technologies CyBOK Knowledge Area [66] and Software Security CyBOK Knowledge Area [9]) is a general approach to verifying properties of systems at runtime [23]. In this approach, one specifies the property ϕ that a system should satisfy and uses a tool to verify that ϕ holds for the actual behaviour(s) observed during execution. The tool may work by running alongside the system and checking that its observed behaviour agrees with the property, or the checks may even be woven into the system itself to be checked as the system executes [69]. When the system fails to satisfy the property ϕ, a relevant part of the current execution trace may be output as a counterexample, witnessing ϕ's failure.

In more detail, runtime verification is often implemented by a monitoring program. This program observes the behaviour of a target system, which is a finite, evolving trace $\pi = s_0 \cdots s_{n-1}$ at some level of abstraction, e.g., the program's inputs and outputs, or perhaps

[2] Taint analysis is also implemented in some tools as a dynamic analysis technique where taint information is tracked at runtime.

(parts of) its internal states. Each new event s_n produced by the system extends the trace π to $s_0 \cdots s_n$ and the monitor determines whether this extended trace satisfies or violates ϕ, and violations are output to the user. Depending on ϕ's structure, violations can be determined immediately once they occur (e.g., for formulas in past-time temporal logics). Alternatively, for other formulas (e.g., those involving future-time operators), the monitor may not be able to report a violation caused by the current event s_n until some time in the future.

Runtime verification has trade-offs compared with other techniques like model checking and theorem proving. Rather than checking whether *all* system behaviours satisfy the desired property ϕ, one checks just that the observed behaviour satisfies ϕ. In this sense, runtime verification amounts to *model checking a trace*. Unlike the previously discussed verification approaches, runtime verification methods produce relatively weak verification guarantees: they can only establish ϕ for the executions they witness. However, in contrast to model checking and theorem proving, these are guarantees with respect to the actual system executing in its real environment, rather than with respect to some model of the system and its environment. Moreover, at least for safety properties, runtime verification does not produce false positives like static analysis does. Finally, the methods used are completely automated and often efficient and scalable in practice.

Recently, progress has been made on runtime monitoring techniques for hyperproperties [70]. In this setting, one may need to give up some of the advantages of runtime verification for trace properties. In particular, since the monitor observes just one trace, but a hyperproperty refers to sets of traces, the monitor may have false positives due to the need to approximate the other unobserved traces.

Runtime verification has been successfully deployed in numerous safety and security-critical settings. NASA is a case in point, where different approaches to runtime verification have been used for the real-time monitoring of Java programs [71]. [72, 73] provide examples of using the MonPoly tool [74] to monitor security policies and data protection policies of distributed systems.

3 HARDWARE

[75, 76, 77, 78]

Hardware security and attacks on hardware were presented in the Hardware Security CyBOK Knowledge Area [79]. Formal methods are now widely used when developing hardware [75] and are indispensable for assurance. As explained in the Hardware Security CyBOK Knowledge Area [79], the Common Criteria standard can be used to guide hardware security evaluations, and its higher Evaluation Assurance Levels mandate the use of formal models and formal methods in this process. More generally, hardware verification is a well-developed research area, see for example the surveys [80, 81]. We highlight here some security-relevant topics in this field.

3.1 Hardware Verification

Microprocessors and other hardware components can be formulated at different levels of abstraction ranging from high-level Hardware Description Languages (HDLs) at the register transfer level (e.g., in HDLs like Verilog or VHDL) to low-level descriptions at the transistor level. Once the description languages used are given a semantics, one may use decision procedures and theorem provers for property verification.

Historically, much of the progress in decision procedures and model checking was driven by hardware verification problems. For example, equivalence checking for combinational circuits led to advances in propositional satisfiability procedures, e.g., data structures like ordered binary decision diagrams [60]. Verifying temporal properties of sequential circuits was a central motivation behind the development of efficient model checking algorithms. Moreover, by giving HDLs a semantics in a language for which verification tools exist, one can directly utilise existing tools for those languages. For example, [82] shows how designs in the Verilog HDL can be translated into equivalent ANSI-C programs. The property of interest is later instrumented into the C program as an assertion. Standard verification techniques can be used to validate the assertion such as bounded model checking, path-based symbolic execution, and abstract interpretation.

For large-scale hardware systems, some form of semi-automated or interactive theorem proving is usually required. For example, ACL2 has been used to verify a number of microprocessors due to its support for automated inductive reasoning (useful to establish invariants about system behaviour) and because designs can be efficiently executed (useful both for simulation and for proofs). The ACL2 verification of a microprocessor is described in [83], which features complex control mechanisms such as out-of-order issue and completion of instructions, speculative execution with branch prediction, and memory optimisation such as load-bypassing and load-forwarding. Such verification can be part of a Common Criteria certification. For example, [84] reports on the certification of the AAMP7G Microprocessor, where the ACL2 theorem prover was used to verify that its microcode, implementing a separation microkernel, was in accordance with EAL 7 requirements (the highest evaluation level in the Common Criteria) and guarantees security-relevant aspects such as space partitioning.

3.2 Side-Channels

Particularly relevant from the standpoint of hardware security is the detection and removal of side-channels. A side-channel is an unintended communication channel where information presumed secret, such as cryptographic keys, can be leaked. Examples of hardware side-channels are presented in the Hardware Security CyBOK Knowledge Area [79], including side-channels and attacks based on timing and power analysis.

Abstractly, detecting side-channels requires analysing hardware (or programs) to determine if one can observe differences in behaviour, depending on values of secret data. This amounts to checking noninterference of secret values on observable outputs, where the observations can include measurements like timing or power usage. Note that analysis of noninterference and other secure information flow notions are typically studied for software (see Section 5.1 for more on this) but some techniques are cross-cutting and have also been applied to hardware. One challenge in reasoning about noninterference for both hardware and software is that, even in well-designed and well-implemented algorithms, often some information can leak. For instance, the failure to decrypt a message or the failure to log-in with a guessed password reveals some limited information about the secret, namely that the guessed key or password

was incorrect. One option here is to explicitly specify where information is allowed to leak in controlled ways. This is called *declassification* [85]. Another option, which we explore further below, is to measure *how much* information can be leaked through a side-channel.

There is a large body of work on side-channel analysis, focusing on foundations and analysis methods as well as attacks and countermeasures. Early research, like [86, 87], developed generic models and frameworks for proving that hardware implementations are free of side-channels, for measuring the amount of information that can be leaked when side-channels exist, and for exploring possible countermeasures to eliminate leakages. As an example, [88] proposes a mathematical model, based on abstract virtual-memory computers, that can be used to develop provably secure cryptography in the presence of bounded side-channel leakage. This model was further specialised into a framework for evaluating side-channel attacks by [76], which measures information-theoretic leakage with respect to adversaries who interact with the system in *non-adaptive* ways.

While the above research focused more on foundations, subsequent research moved closer to tool-supported formal methods. In [89], Köpf et. al. propose a framework that, like [76], uses information-theoretic metrics to measure side-channel leakage. However in this work, the metrics are used to quantify the information revealed to an *adaptive* adversary who can make timing or power measurements, based on the number of interactions with the system under attack. The essential idea is to search over all adversary attack strategies, which are the adaptive decisions that the adversary can make, and use information-theoretic entropy measures to express the adversary's expected uncertainty about the secret after interacting with the system using each strategy. This framework is instantiated with a hardware description environment, thereby supporting the automated detection and quantification of information leakages in hardware algorithms, like those implementing cryptography.

More recent research incorporates ideas from program analysis to track the observations that can be made by adversaries. For example, for reasoning about micro-architectural side-channels, the cache audit tool [77] uses an abstract interpretation framework, with a suitable abstract domain, to track information about the possible cache states. This is used to quantify the information leakage possible by measuring the timings associated with cache hits and misses that can occur during execution on different control flow paths. The tool takes as input a program binary and a cache configuration and derives formal, quantitative security guarantees against adversaries that can observe cache states, traces of hits and misses, and execution times.

Researchers have also developed techniques that use formal models and program analysis to either *generate* programs that are, guaranteed to execute in constant time (independent of secret data) or *repair* programs with timing leaks by ensuring that all branches take the same time [90, 91, 92, 93]. There is, however, still considerable research left to do. Most current work stops at the level of intermediate representation languages, rather than going all the way down to machine code. Another deficit lies in the fact that the models are not detailed enough to say what happens under speculative or out-of-order execution at the hardware level. Overcoming these limitations remains an important challenge.

3.3 API Attacks on Security Hardware

Even when hardware implements the ”right” functionality and is free of side-channels, it may still be vulnerable to attacks that leak secrets. An active research area has been on attacking security hardware by abusing its API, effectively by sending the hardware valid commands but in an unanticipated sequence that violates the designer's intended security policy [78]. Such attacks have been carried out against a wide variety of security-critical hardware including cryptographic co-processors and more specialised processors designed for automated teller machines, pay-TV, and utility metering. The problem of API attacks is cross-cutting as they are also applicable to software libraries. API attacks can also be seen as a protocol problem in how the API is used; hence formal methods and tools for security protocols, described in Section 4, are relevant for their analysis.

One particularly successful application of formal methods has been the use of protocol model checkers to detect attacks on security tokens over their RSA PKCS#11 APIs. As [94] shows, given a token it is possible to reverse-engineer a model of its API in an automated way. This model can then be input to a security protocol model checker to search for attacks where sensitive keys are exposed to the adversary.

4 CRYPTOGRAPHIC PROTOCOLS

[95, 96, 97, 98, 99, 100]

Security protocols are a superb showcase for formal methods and how their use can improve the security of critical systems and their components. As noted in [99], security protocols play a role analogous to fruit flies for genetic research: they are small and seemingly simple. However, for protocols, this simplicity is deceptive as they are easy to get wrong. A classic example of this is the attack that Lowe found in the 1990s on a simple three step entity authentication protocol that is part of the Needham Schroeder Public Key protocol [98].

Formal methods for security protocols have been the subject of considerable research for over four decades; see [101] for a survey on security protocol model checking and [102] for a more general survey on computer-aided cryptography. Our focus in this section is primarily on using formal methods to analyse security protocol *designs*; we return to *implementations*, in particular of cryptography, in Section 5.2. Design verification has high practical relevance as if designs are not secure, then no (design-conform) implementation will be secure either. In the past, this issue was not taken seriously enough and most protocols were standardised without a clear formulation of either their properties or the intended adversary [103]. This situation is now improving, albeit slowly. Protocol verification tools are having a practical impact on the design of security protocols that matter, such as ISO/IEC Protocols for Entity Authentication [104], TLS 1.3 [105, 106, 107], 5G [108], and the EMV standard for credit-card payments [109, 110].

4.1 Symbolic Methods

Symbolic models are the basis of many successful formal methods approaches to reasoning about protocols and applications that use cryptography. In the symbolic setting, messages are represented by terms in a term algebra containing cryptographic operators. The adversary is modelled as being active: he controls all network traffic and can manipulate terms, e.g., he can concatenate terms together or decompose concatenated terms into their subterms. Properties of cryptography are typically formalised with equations or inference rules. For example, for symmetric encryption the equation

$$decrypt(encrypt(m, k), k) = m$$

would formalise that all parties (including the adversary) can decrypt any message m encrypted with a key k, by using the same key for decryption. Cryptography is assumed to work perfectly, in an idealised way, in that the adversary cannot do anything more than is specified by such equations. So either the adversary can decrypt a cipher text $c = encrypt(m, k)$ yielding the plain text message m, if he possesses the right decryption key k, or no information about the plain text is leaked. This kind of model of an active network adversary who has access to all message terms, can manipulate them, but cannot "break" cryptography is sometimes called the "Dolev-Yao adversary" after the seminal work of Dolev and Yao [111].

A security protocol is specified by a set of parameterised processes, called *roles*, that describe the actions taken by the agents executing the role. For example, in a key agreement protocol there might be an initiator role, a responder role, and a key-server role. Protocols are given an operational semantics where agents may play in multiple roles. For example, Alice may be the initiator in one protocol run and a responder in a second run. So in general there may be arbitrarily many role instances. This setup gives rise to a transition-system semantics with an associated notion of trace. The transition system is given by the parallel composition of unboundedly many role instances together with a process describing the Dolev-Yao adversary. The transition system is infinite-state as there can be arbitrarily many processes. Also note that the adversary himself is an infinite-state process; this reflects that he is very prolific in that he can produce and send infinitely many different messages to the other agents running in the different roles. For example, if the adversary has seen a message m, he can always concatenate it with itself (or other messages he has seen) arbitrarily many times, hash it, repeatedly encrypt it, etc. Finally, security definitions are formulated in terms of trace properties, which are often simple invariants. For example, the adversary never learns a key presumed to be secret.

4.1.1 Theorem Proving

One way to establish a safety property P about the traces of an infinite-state transition system is to show that P is inductive: it holds for the empty trace and, assuming it holds for an arbitrary (finite) trace π, it holds for all extensions of π by an event s, i.e., $P(\pi\, s)$ follows from $P(\pi)$.

This is the essence of the approach taken by Paulson [97] where protocols and the actions of the Dolev-Yao adversary are encoded as inductive definitions in higher-order logic. The properties of protocols are also formalised in higher-order logic as properties of finite traces.[3]

[3] The restriction to finite traces is necessary as protocols are defined inductively as sets of finite traces (rather than co-inductively as infinite traces). However, this is not a restriction in practice, unless one wishes to reason about liveness properties, which is rarely the case for security protocols.

For example, a common class of properties is that one event s always precedes another event s'. This is relevant for authentication and agreement properties [112], for example, when Bob finishes a protocol run in his role then Alice previously finished in her role. Another important class of properties are secrecy properties, e.g., that a session key established in a key exchange protocol is secret even after the subsequent compromise of any long-term keys (perfect forward secrecy).

Given a specification of a protocol, including the adversary and the protocol's intended properties, Paulson then proves by induction, for each property P, that every trace in the protocol's inductive definition satisfies P. The inductive proofs are constructed using Isabelle/HOL [45]. Theorems are proven interactively, which may require considerable work, say to establish the auxiliary invariants needed for proofs. [97] reports that the time needed for an expert to construct proofs ranged from several days for small academic protocols to several weeks for a protocol like the handshake of TLS v1.3. Despite this effort, numerous nontrivial protocol were proven this way, such as the TLS v1.3 handshake, Kerberos v1.4, and sub-protocols of SET (Secure Electronic Transactions [113]).

This inductive method was further refined by [114, 115], who use derived proof rules to mimic the backwards search from an attack state to an initial state. Proofs are again constructed in Isabelle/HOL, but the proof rules also support substantial automation. The method reduces the time required for proofs in comparison to [97] by several orders of magnitude, whereby the security of "academic" protocols can often be proven completely automatically.

In contrast to post-hoc verification, one can use theorem provers to *develop* protocols hand-in-hand with their correctness proofs using step-wise refinement. Refinement [116, 117] allows one to decompose the complexity of verifying properties of a complex (transition) system into verifying the properties of much simpler, more abstract systems, and establishing refinement relationships between them. Namely, one starts with a very abstract transition system model M_0 of the desired (distributed) system and refines it into a sequence of increasingly concrete models $M_1, M_2, \ldots M_n$, proving properties of the system at the highest possible level of abstraction, and establishing a refinement relation between each pair of models M_i and M_{i+1}, for $i \in \{0, \ldots n-1\}$. The existence of transitive refinement relations ensure that properties proven for the more abstract models also hold for the more concrete models.

The refinement approach has been applied to security protocols in [118, 119], where a refinement strategy is given that transforms abstract security goals about secrecy and authentication into protocols that are secure when operating in an environment controlled by a Dolev-Yao-style adversary. This approach simplifies the proofs of properties and provides insights on why protocols are secure via the proven invariants. Furthermore, since a model may be refined in multiple ways, the refinement steps can be structured as a tree, rather than as a sequence. This fosters the development of families of protocols, given by the models at the leaves of the tree, which share common structure and properties. The refinement method is implemented in Isabelle/HOL and used, for example, to develop a family of entity authentication and key establishment protocols that include practically relevant features such as key confirmation, replay caches, and encrypted tickets.

4.1.2 Model Checking Trace Properties

Despite the undecidability of the underlying question of whether a protocol is secure in the symbolic model [120], it is still possible to build effective analysis tools. Early tools, such as Millen's Interrogator [121], the Longley-Rigby search tool [122], and the NRL Protocol Analyzer [123], were search procedures that exhaustively searched the problem space or some part of it. While this would normally not provide correctness guarantees, these tools were still very effective in finding subtle attacks on security protocols.

One way to approach model checking is to reduce it to a problem that can be tackled using standard algorithmic-verification tools. We give two examples of this. The first is exemplified by the use of CSP and FDR [99]. CSP is a general language and theory for modelling systems consisting of interacting processes and FDR (and its more recent incarnations like FDR2) is a model checker for CSP. The basic idea is that the system analysed consists of processes describing the different protocol roles which are run in parallel with a process describing the adversary. These processes are either formalised directly in CSP or, compiled from a more abstract description [124]. The security property is also formalised as a CSP process. FDR is then used to establish, via trace refinement, that the language of the system is contained in the language of the property. Since the FDR tool only handles processes with finite alphabets and finite state spaces, these must be bounded, e.g., by bounding the number of role instances and the size of messages that agents and the adversary can generate. This is a practical limitation since by introducing bounds, one may miss attacks.

A second example of using standard deductive tools is the SATMC model checker [125]. SATMC takes as input a protocol and property description, given in a high-level input language [126], and translates them into a propositional logic satisfiability problem, essentially encoding a bounded reachability (or planning) problem: can the adversary drive the protocol into an attack state? This is then solved using an off-the-shelf SAT solver. As with the use of CSP and FDR, the main challenge and limitation is the restriction to a finite state space.

More recent state-of-the-art tools, such as ProVerif [96, 39] and Tamarin [100, 127], use specialised algorithms to efficiently manage and search the infinite state space defined by the protocol and adversary. Both tools support user-defined cryptographic primitives specified by equations and the verification of trace properties and privacy properties, the latter specified using observational equivalence. ProVerif takes as input a protocol description in a process calculus, called the applied pi calculus, cf.Section 2.1.3. It then translates the protocol description into a set of Horn clauses, applying domain specific approximations and abstractions in the process. For example, individual fresh values (used for modelling secrets) are abstracted into sets of fresh values and each action in a process can be executed multiple times. Given the abstracted protocol model, ProVerif uses resolution to determine whether relevant properties hold; these include secrecy and correspondence properties [128] or observational equivalence [129]. The ProVerif tool is widely used and is also part of other tool chains. For example, it is used in [130] to analyse protocol implementations written in F#, which can be directly executed.

In Tamarin, protocols are specified using multiset rewriting rules and properties are specified in a fragment of first-order logic that supports specifications over traces. Proofs are constructed by a backwards search using constraint solving to perform an exhaustive, symbolic search for executions with satisfying traces. The states of the search are constraint systems and one starts with a system specifying the negation of the desired properties; hence one uses Tamarin to systematically find attacks. More generally, a constraint can express that some multiset rewriting step occurs in an execution or that one step occurs before another step.

Formulas can also be used as constraints to express that some behaviour does not occur in an execution. Applications of constraint reduction rules, such as simplifications or case distinctions, correspond to the incremental construction of a satisfying trace.

Tamarin's constraint reduction rules are sound and complete. This means that if one arrives at contradictions in all leaves of the derivation, then no satisfying trace exists; so we have a proof that there is no counterexample to the desired property, i.e, the property holds. Alternatively, if no further rule can be applied, but there is no contradiction in a leaf, then we can construct a trace that represents a counterexample to the desired property, i.e., we have an attack. Rule application can be carried out automatically or with user guidance, which may be necessary when automated rule application fails to terminate. Tamarin has special built-in support for equational theories relevant to security protocols, such as Diffie-Hellman exponentiation. Tamarin also supports, in contrast to ProVerif, the specification of protocols with persistent global state, e.g., counters, memory cells, etc., that are shared between protocol participants. Finally, Tamarin's specification language can be used to specify a wide variety of adversary models in the symbolic setting [131].

4.1.3 Model Checking Non-trace Properties

As described in Sections 2.1.2 and 2.1.3, privacy-style properties are typically not trace properties, but can be expressed as hyperproperties or as observational equivalences between processes. Several model checkers support proving limited kinds of equivalences between protocols.

In the ProVerif model checker, one can formulate and prove a special kind of equivalence called *diff-equivalence* between processes [132, 38, 133]. Diff-equivalence is a property that is stronger than observational equivalence and hence proofs of diff-equivalence suffice to establish observational equivalence and can be used to verify many privacy-style properties. The key insight is that diff-equivalence limits the structural differences between processes, which simplifies automating equivalence proofs. In more detail: two processes, P and Q, are specified by a new kind of process B, called a *biprocess*, which is a process in the applied pi calculus except that it may contain subterms of the form *diff*$[M, M']$ (where M and M' are terms or expressions) called *diff-terms*. P is then the process, where each diff-term in B is replaced by its first projection (e.g., M in the above example). Similarly Q is the process constructed by replacing each diff-term with its second projection (e.g., M'). Hence P and Q have the same structure but differ just in those places distinguished by a diff-term in B. ProVerif contains support to automatically establish the diff-equivalence of P and Q. Both Tamarin [134] and Maude-NPA [135] have subsequently also added support to verify variants of diff-equivalence.

Static equivalence [38] is another property, related to indistinguishability, which is supported by different tools. Static equivalence is defined with respect to an underlying equational theory. It states, roughly speaking, that two terms are statically equivalent when they satisfy the same equations. This essentially amounts to a special case of observational equivalence that does not allow for the continued interaction between a system and an observer: the observer gets data once and conducts experiments on its own. Decision procedures for static equivalence have been implemented by tools including YAPA [136], KISS [137], and FAST [138].

Static equivalence is useful for a variety of modelling problems in computer security. For example, it can be used to model off-line guessing attacks, where the adversary tries to guess a secret and verify his guess, without further communication with the system. This problem

is studied in [139, 140], who formulate the absence of off-line guessing attacks by using static equivalence to express that the adversary cannot distinguish between two versions of the same symbolic trace: one corresponding to a correct guess and the other corresponding to an incorrect guess. Decision procedures, like those listed above, are then used to answer this question.

4.2 Stochastic Methods

In the symbolic setting, there are no probabilities, only non-determinism, and security definitions are possibilistic (the adversary cannot do something bad) rather than probabilistic (he can do something bad only with negligible probability). Probabilities, however, are part of standard cryptographic definitions of security, cf. Section 4.3. They also arise when protocols are based on randomised algorithms or the guarantees themselves are probabilistic. Different options exist for augmenting both transition systems [141] and logics with probabilities, and the corresponding model checking algorithms combine methods from conventional model checking, methods for numerical linear algebra, and standard methods for Markov chains [142].

An example of a popular and powerful model checking tool is PRISM [65, 143, 144], which supports the specification and analysis of a different type of probabilistic models. These include discrete and continuous-time Markov chains, Markov decision processes, and probabilistic automata and timed variants thereof. PRISM supports different property specification languages including Probabilistic Computation Tree Logic (PCTL) and Continuous Stochastic Logic (CSL), both based on the temporal logic CTL.

Probabilistic model checking has been successfully applied to a variety of protocols for security and privacy, where the protocols incorporate randomisation and the properties are probabilistic. For example, [145] used PRISM to model the protocol underlying the Crowds anonymity system [146], which is a peer-to-peer protocol for anonymous group communication based on random message routing among members of a “crowd”. [145] models the behaviour of the group members and the adversary as a discrete-time Markov chain and the system’s anonymity properties are formalised in PCTL. PRISM is used to analyse the system, showing, for example, how certain forms of probabilistic anonymity change as the group size increases or random routing paths are rebuilt. A second example is the use of PRISM to analyse a probabilistic model of a randomised non-repudiation protocol that guarantees fairness without resorting to a trusted third party [147]. Here the model checker is used to estimate the probability of a malicious user breaking the non-repudiation property, as a function of different protocol parameters. Two additional examples are the use of PRISM to analyse the probabilistic fairness guarantees provided by a randomised contract signing protocol [148] and the probabilistic modeling and analysis of security-performance trade-offs in Software Defined Networking [149].

In Section 2.1.2 we discussed the relevance of hyperproperties for security. Model checkers have also been developed to support the verification of such properties. Namely, hyperproperties are specified in the logics HyperLTL or HyperCTL* and automata-based algorithms are used to check finite-state systems with respect to these specifications [28, 29]. Recent work has also shown how to specify and reason about hyperproperties in the context of Markov Decision Processes [150, 151]. For example, in [150], a specification language is given called Probabilistic Hyper Logic that combines a classic probabilistic logic with quantification over schedulers and traces. This logic can express a variety of security-relevant properties such as

probabilistic noninterference and even differential privacy. Although the model checking problem is undecidable in this case, it is nevertheless possible to provide methods for both proving and refuting formulas in relevant fragments of the logic that include many hyperproperties of practical interest.

4.3 Computational Methods

Cryptography has a history going back thousands of years. However, until the last century it was approached more as an art than a science. This changed with the development of public key cryptography and the idea of *provable security*, where reduction arguments are used to put the security of cryptographic schemes on a firm scientific footing. In contrast to symbolic models, the definitions and proofs involved are at a much lower level of abstraction, sometimes called the *computational* model. In this model, agents manipulate bit strings rather than terms and the adversary's capabilities are modelled by a probabilistic polynomial-time Turing machine rather than a process in a process calculus or an inductive definition in higher-order logic. Additionally, security definitions are probabilistic rather than possibilistic. For example, they are formulated in terms of the adversary's chances of winning some game, cf. Cryptography CyBOK Knowledge Area [152].

There is a trade-off between computational and symbolic models: the computational models are more detailed and accurate, but substantially more effort is involved in formalising protocols and in proving their properties. This trade-off is not surprising. In fact it is standard from both the formal methods and the security perspective: concrete models provide a more precise description of how a system works and what the adversary can do, but formalisation and proof construction is correspondingly more difficult. The difficulty in applying these ideas in practice is substantial and some researchers have even questioned whether the gains are worth it or, given the complexity, whether one can trust the results. [153] contains a brief history of the topic and surveys some of the past controversies around this question.

To increase confidence in the validity of cryptographic proofs in the computational setting, various proposals have been made. For instance different structuring techniques and abstractions were proposed to simplify constructing proofs and understanding the resulting security arguments. Prominent examples are game-based proofs [154, 155], Canetti's universal composability framework [156], and Maurer's constructive cryptography [157]. In addition, in response to Halevi's call [158] for increased rigour in cryptographic proofs, formal-methods tools were developed to help cryptographers make the transition from pen-and-paper proofs to mechanically checked security proofs in the computational setting. There has also been some success in bridging symbolic models and computational models via computational soundness results for symbolic abstractions [159, 160, 161, 162, 163]; these results enable symbolic proofs, but with stronger, computational guarantees.

In the following, we expand on representative approaches and associated tools for carrying out game-based and simulation-based proofs.

4.3.1 Game-based Proofs

Games and game transformations can be used to structure cryptographic proofs. A game describes the interaction between different parties (typically the adversary and some other entity), which are modelled as probabilistic processes. Proofs are structured as a sequence of games where the initial game represents the security goal (for example, formalising indistinguishability under chosen-ciphertext attacks) and the final game is one where the adversary's advantage is zero by inspection. The transitions between games are based on small transformations that preserve, or only slightly alter, the overall security, e.g., by transforming one expression into an equivalent one or applying a transformation based on some computational hardness assumption (cf. Cryptography CyBOK Knowledge Area [152]). This is expressed in terms of the probability of events occurring in each of the games; for example, the probability of an event A happening in a game G_i is close to the probability of some event A' happening in game G_{i+1}. A good overview and tutorial on constructing such proofs is [155].

CryptoVerif [164, 95] was the first tool developed to support game-based proofs. In CryptoVerif, games are modelled as processes in a language inspired by the pi calculus and transitions are justified by process-algebraic notions like bisimulations. The tool can prove secrecy and correspondence properties, which are relevant, for example, for authentication and key agreement protocols. Moreover, CryptoVerif is highly automated: the tool can automatically decompose games into reductions and oracles and even discover intermediate games, rather than having them given explicitly by the user. To achieve this, CryptoVerif imposes various restrictions. For example, the game transformation strategies are hard-coded and users must trust that CryptoVerif correctly implements them and that the transformations' pen-and-paper justifications are correct. The tool also lacks extensibility as it supports just a fixed set of language primitives and game transformations. Nevertheless, CryptoVerif is quite powerful and has been applied to both protocols and primitives, including Kerberos, and the Full-Domain Hash signature scheme.

An alternative approach to mechanising game-based proofs is to build the foundations for these proofs directly within an expressive logic like a type theory (for example, the calculus of inductive constructions) or higher-order logic. This is the approach taken by tools like CertiCrypt [165] and FCF [166], which are implemented in Coq, and CryptHol [167], implemented in Isabelle/HOL. These approaches are considered *foundational* in that they start with a formalised semantics. In particular, the tools provide a language for probabilistic programs with a semantics formalised within the logic. On top of this, one can

express relevant concepts for game-based proofs like discrete probability distributions, failure events, games, reductions, oracles, and adversaries. The semantics is used to formally derive proof rules for reasoning about probabilistic programs and game transformations, e.g., to replace subprograms by equivalent subprograms. Deriving these rules ensures the consistency of the resulting theory and one thereby has stronger guarantees than is possible by directly axiomatising proof rules. Moreover, the underlying logic (e.g., HOL) can directly be used to formulate and prove the correctness of arbitrary games and transformations between them. Examples of the kinds of theorems proven using the above tools are the semantic security of OAEP (with a bound that improves upon existing published results) and a proof of the existential unforgeability of FDH signatures (in CertiCrypt), the security of an efficient scheme for searchable semantic encryption (in FCF), and the IND-CCA security argument for a symmetric encryption scheme (in CryptHol).

Proof construction using these tools usually requires significant manual effort since the user has a free hand in structuring games, their transformations, and justifications. The EasyCrypt

tool [168] shows, however, that some automation is possible by using appropriate programming logics and deductive machinery. EasyCrypt provides a language for formalising the probabilistic programs used to model security goals and hardness assumptions, as well as a probabilistic Hoare logic and a relational version thereof for reasoning about procedures (games) and pairs of procedures (game transformations). Proof construction can be automated using tactics, which are programs that build proofs, along with SMT solvers. EasyCrypt users interact with the system by providing proof sketches for game-hopping proofs and the above deductive machinery is then used to construct the resulting proofs.

4.3.2 Simulation-based Proofs

Simulation-based security proofs are based on a general paradigm for formalising security definitions and establishing the correctness of systems with respect to them. The main idea is that the security of a real system (or protocol) is abstractly defined in terms of an ideal system, otherwise known as an ideal functionality. For example, semantic security for encryption could be formalised by comparing what an adversary can learn when receiving a real ciphertext with what the adversary can learn when receiving just a random value. A real system securely realises the ideal system if every attack on it can be translated into an "equivalent" attack on the ideal system. Here, the notion of equivalence is specified based on the environment that tries to distinguish the real attack from the ideal one. This approach is attractive due to the general way that security can be defined and also because it yields strong composability properties. Security is preserved even if the systems are used as components of an arbitrary (polynomially bounded) distributed system. This supports the modular design and analysis of secure systems. A good overview and a tutorial on constructing such proofs is [169].

Numerous proposals have been made for formalizing simulation-based security, see e.g., [170], all with different scopes and properties. These include universal composability [156], black box simulatability [171], inexhaustible interactive Turing machines [172], constructive cryptography [157], and notions found in process calculi, both with [173] and without [174] probabilities. Support for formalising proofs in these frameworks is, however, not yet as advanced as it is for game-based proofs and progress has been more modest. For example, [175] formalised a computationally-sound symbolic version of blackbox simulatability in Isabelle/HOL and used it to verify simple authentication protocols. [176, 177] have carried out simulation proofs in Isabelle/HOL, building on CryptHOL, for example verifying multi-party computation protocols. There has also been some progress on automating simulation-based proofs using the EasyCrypt system to mechanise protocol security proofs within a variant of the universally composable security framework [178].

5 SOFTWARE AND LARGE-SCALE SYSTEMS

[179, 180, 181, 182, 183, 184, 185]

There are many well-established methods and tools for verifying functional properties of programs, for example, verifying that a program has a given input/output relation or satisfies a given trace property. As explained in Section 2.1, this is relevant for security. Also of direct relevance are the numerous tools employing both sound and unsound analysis procedures that can help identify common security problems, such as memory corruptions, injection attacks and race conditions. We refer to [186] for a general survey of formal methods techniques, [187] for a survey on automated methods, and to [188] for an excellent overview of some of the

```
h := h mod 2
l := 0
if (h = 1)
  then l := 1
  else skip
```

Figure 1: Example of Information Flow

successes and challenges in applying formal methods to large-scale systems like operating systems and distributed systems.

In the following subsections we examine methods, tools, and applications in the context of general software, cryptographic libraries, and other kinds of systems. We start with information flow control, as it provides a foundation for enforcing end-to-end security policies.

5.1 Information Flow Control

Information flow control (IFC) concerns the enforcement of confidentiality or integrity guarantees during a system's execution. The basic setup distinguishes different security levels, such as high and low or, more generally, a lattice of levels. For confidentiality policies, information should not flow from high to low and dually for integrity properties, cf. the Biba model from the Operating Systems & Virtualisation CyBOK Knowledge Area [189] or taint tracking, where information should not flow from tainted to untainted. As explained in Section 2.1.2, noninterference and related hyperproperties can often be used to formulate such policies.

Figure 1, taken from [179], contains a simple example of a program that violates a secure information flow policy for confidentiality that prohibits information from the high variable h from flowing to the low variable l. Due to the if-branching in this example there is an implicit flow where the value of h may flow to l.

Information flow control is of particular importance for program and system verification addressing *end-to-end* security guarantees: information flows must be restricted throughout the entire system, from start to finish, to prevent violations of confidentiality or integrity policies. Real-world programs are of course much more complex than this and it may not be so trivial to determine whether the high variable affects the value of low variables anywhere throughout the program. When the property holds, the guarantee is a strong one. This is in contrast to what is normally achieved by conventional security mechanisms like access control, which controls the context in which data can be released, but not how the data is used after its release.

Below we give examples of popular approaches to verifying that programs ensure secure information flow in practice.

5.1.1 Static Analysis and Typing

Early work on IFC focused on preventing information flows using static analysis [190] and specialised typing disciplines [191]. Specifically, in [191], a type system was developed that soundly enforces a secure information flow property inspired by noninterference. The rough idea is that each program expression has a type, which includes not just a conventional type, like Int or Bool, but also a security label. Typing rules associate types with expressions and make statements about expressions being well typed based on the types of their subexpressions. For example, in a lattice with just the labels high and low, any expression can have the type high, whereas it has type low if all its variables are associated with the type low. Moreover, an expression e can only be assigned to a low variable l, representing a direct information flow, if e has type low. The typing rules also prevent indirect information flows by making sure that low variables cannot be updated in high contexts. For example, an if-then-else expression branching on a Boolean b would allow low assignments in its branches only if b is low typeable. This rules out programs like that in Figure 1. For further details see, e.g., [192, 179].

Various languages and compilers now support security type systems. These include Jif (Java + Information Flow) [180, 193], Flow Caml [194, 195], which is an information flow analyser for the Caml language, the SPARK programming language built on top of Ada [196] (based in part on the seminal work of [197]), and JOANA for Java [198]. For example, Jif extends Java with security types to support information flow control and access control, enforced both statically at compile time and at run time. In Jif, the variable declaration

$$\text{int } \{\text{Alice} \rightarrow \text{Bob}\}\ x;$$

expresses both that x is an integer and that Alice requires that the information in x can only be seen by Bob. In contrast,

$$\text{int } \{\text{Alice} \leftarrow \text{Bob}\}\ x;$$

expresses Alice's requirement that information in x can only be influenced by Bob, which is an integrity policy. The Jif compiler analyses these labels to determine whether the given Java program respects them.

The research of [198] illustrates how richer secure information flow properties can be handled in full Java with concurrency. Traditional information flow control focuses on possibilistic leaks: can secret (high) data directly or indirectly leak to non-secret (low) data. However, this is insufficient when programs have parts that may execute concurrently since the scheduling of their threads can effect which values are output or their ordering. [198] investigates probabilistic noninterference, which is relevant for concurrent programs as it accounts for probabilistic leaks where, for some scheduling strategies, the probability of some publicly visible behaviour depends on secret data. In particular, [198] develops sophisticated program analysis techniques to enforce a version of probabilistic noninterference, called *low-security observational determinism*, which requires that execution order conflicts between low events are disallowed if they may be influenced by high events. The resulting tool, JOANA, takes Java programs, where all inputs and outputs are labelled high or low and determines whether this property holds. JOANA can handle full Java byte code, it scales to programs with an arbitrary number of threads and 50k+ lines of code, and its (conservative) analysis is of high precision.

5.1.2 Self-composition and Product Programs

As explained in Section 2.1.2, information flow properties are hyperproperties that relate pairs of executions of a program P. A natural idea, rather than reasoning about pairs of executions, is to reason about a *single* execution of a program P', derived by composing P with a renaming of itself [199]. Depending on the programming language and the properties involved, this self-composition may be just the sequential composition of programs. Namely P' is simply $P; R(P)$, where $R(P)$ is a version of P with renamed variable. Alternatively P' may be a more complex product construction, for example, based on P's transition system representation [200].

This idea is powerful and has wide scope. For example, one can systematically reduce different notions of secure information flow for a program P to establishing a safety property for the self-composition of P. Furthermore, the idea generalises beyond secure information flow: hyperproperties covering both safety and liveness can be formalised and reasoned about using forms of self-composition [201, 14].

From the deductive perspective, a strong advantage of both self-composition and product programs is that one can reuse off-the-shelf verifiers to reason about the resulting program. This includes using weakest-precondition calculi and Hoare logics [202, 203], separation logics, and temporal logics. When the program is finite-state, temporal logic model checkers can also be employed as decision procedures. Indeed, self-composition provides a basis for LTL model checking for hyperproperties [28]. Note that an alternative proposal is not to combine programs but to use a specialised *relational logic* to explicitly reason about pairs of programs [204, 205].

Overall, the above ideas have seen wide applicability: from reasoning about secure information flow to detecting, or proving the absence of, timing and other side-channels, cf. the example with timed traces in Section 2.1.2. An example of the practical application of these ideas is [181]. The authors present a state-of-the-art theorem proving environment for realistic programming languages and extend it to support the verification of arbitrary safety hyperproperties, including secure information flow. The key idea is a specialised product construction that supports the specification of procedures and the modular (compositional) reasoning about procedure calls. This product construction is implemented within the Viper verification framework [206], which is an off-the-shelf verifier supporting different language front-ends (such as Java, Rust, and Python), an intermediate language (that is a simple imperative language with a mutable heap and built-in support for specifications) and automation backends that leverage symbolic execution, verification condition generation, and SMT solvers. The tool has been used to reason about information flow properties, including those involving declassification, of challenging programs. The programs combine complex language features, including mutable state on the heap, arrays, procedure calls, as well as timing and termination channels. User interaction with Viper is mainly limited to adding pre-conditions, post-conditions, and loop invariants and Viper completes the proofs in reasonable time.

5.2 Cryptographic Libraries

Reasoning about the implementations of cryptographic primitives and protocols involves challenges that go beyond the verification of their designs, as discussed in Section 4. Specifically, the methods and tools for code-level verification must work directly with the implementations themselves. Moreover, the properties to be verified must also account for new problems that can arise in the implementation [182], such as the following.

- **Freedom from side-channel attacks:** This includes that secrets are not revealed (secure information flow) by software or hardware artifacts including assignment, branching, memory access patterns, cache behaviour, or power consumption.
- **Memory safety:** Only valid memory locations are read and written, for example there are no dangling pointers that reference memory that has been deallocated.
- **Cryptographic security:** The code for each cryptographic construction implements a function that is secure with respect to some standard (computational or information-theoretic) security definition, possibly under cryptographic assumptions on its building blocks.

Note that for programs written in low-level languages like C or assembler, memory safety is particularly important since a memory error in any part of the code could compromise the secrets it computes with or lead to the adversary taking control of the application. This necessitates the ability to reason effectively about low-level code.

One approach to the verification of cryptographic libraries is that taken by HACL* [182], which is a verified version of the C cryptographic library NaCL. The NaCL library is a minimalistic, but fully-functional library of modern cryptographic primitives including those for encryption, signatures, hashing and MACs. Since C is difficult to reason about, the library is reimplemented in F* [207], which is a functional programming language designed to support program verification. F* features a rich type system and supports dependent types and refinement types, which can be used to specify both functional correctness and security properties. Type checking is mechanised using an SMT solver and manual proofs. The NaCL library is written in a subset of F* that can be translated to C and subsequently to assembler using either conventional or verified compilers. The NaCL library is verified with respect to its functional correctness, memory safety, and the absence of timing side-channels where secrets could be leaked; cryptographic security was not covered however. For example, memory safety is proven by explicitly formulating properties of programs that manipulate heap-allocated data structures. Functional correctness is proven with respect to executable specifications that are themselves checked using test vectors.

An alternative to working with a high-level language like F* is to work with a lower-level language that is an assembly language or reasonably close to one. This eliminates or reduces the need to trust that compilers will respect the properties proven of the high-level code. Moreover, the implementations can be highly optimised, which is often a requirement for cryptography in practice. Two representative examples of this are Vale and Jasmin.

The Vale tool [208] is used to verify cryptographic functions written in a low-level language, also called Vale, designed for expressing and verifying high-performance assembly code. The Vale language is generic, but targets assembly code by providing suitable language constructions such as operations on registers and memory locations. Moreover, Vale programs can be annotated with pre-conditions, post-conditions, assertions, and loop invariants to aid proofs. Vale code is translated into the Dafny logical framework [209] and the operational semantics of

the assembly language, e.g., x86, is written directly in Dafny. Since the operational semantics can account for the timing of operations and memory accesses, one can establish the absence of (i) timing and (ii) cache side-channels by showing that (i) no operations with variable latency depend on secrets and that (ii) there is no secret-dependent memory access, respectively. The Vale tool implements a specialised static analysis procedure to verify this noninterference. Dafny's off-the-shelf verification tool is additionally used to verify functional correctness, where proofs are partially automated using the Z3 SMT solver.

Jasmin [210] (and also CT-Verif [211]) is an alternative approach based on a low-level language for writing portable assembler programs, also called Jasmin, and a compiler for this language. The Jasmin language was designed as a generic assembly language and the compiler platform (currently limited to produce x86_64 assembly) has been verified in Coq. The language and compiler are designed to allow fast implementations, where precise guarantees can be made about the timing of instructions. This enables verifying the absence of timing and cache-based attacks, in addition to functional correctness and memory safety. Verification leverages a tool chain similarly to Vale, using Dafny and the Z3 SMT solver. Jasmin has been used, for example, to verify assembler code for Curve25519, an elliptic-curve variant of Diffie-Hellman key exchange.

Finally, we mention Cryptol[4], an industrial-strength domain-specific language for formalising and reasoning about efficiently-implementable hardware. Cryptol features an executable specification language, a refinement methodology for bridging high-level specifications with low-level implementations, e.g., on FPGAs, verification tools involving SAT and SMT solvers, as well as a verifying compiler that generates code along with a formal proof that the output is functionally equivalent to the input.

5.3 Low-level Code

Cryptographic libraries are just one example of security-critical software, often written in assembler for performance reasons. Operating systems and device drivers typically also contain assembly code and similar verification techniques are needed to those just described. Here, we briefly survey some additional, representative, approaches and tools that have been used to verify low-level code.

As with HACL*, Vale, and Jasmin, a common starting point is a generic low-level language that can be specialised to different platforms, rather than starting with a concrete assembly language like x86. The language chosen can then be given a formal semantics, by embedding it in a higher-order logic like Coq, e.g., as was done with Jasmin. Such an embedding enables one to (1) prove that the translation to assembler preserves desired functional or security properties[5] and (2) derive proof rules for reasoning about low-level programs. For example, [214] builds on separation logic [215], which is a powerful logic for reasoning about imperative programs that manipulate pointers, to build a verification environment for low-level code. Namely, on top of a formalisation of separation logic in Coq, the authors of [214] formalise an assembler and derive proof rules for the verification of assembly code. The Bedrock project [183, 216, 217] provides another Coq framework for verifying low-level code using separation logics. A final example is given by the Igloo project [218], which links a refinement-based formalism for developing programs in Isabelle/HOL with the Viper verification environment,

[4] **https://www.cryptol.net/**

[5] More generally, one may employ techniques for *secure compilation* [212, 213], whereby compilers are guaranteed to preserve security properties of the source program in the target compiled program.

which also supports the verification of low-level code using a separation logic.

In contrast, one may proceed in the reverse direction, by translating assembler into a higher-level programming language for which verification tools already exist. This is the approach taken by Vx86 [219]. It takes as input x86 assembler code that has been manually annotated with verification conditions — pre-conditions, post-conditions, and loop invariants — and translates it to annotated C code, which makes the machine model explicit. The C code is then analysed by VCC, Microsoft's Verifying C Compiler [220], which employs its own verification tool chain involving Z3.

5.4 Operating Systems

Operating systems are critical for security as they provide services that applications rely on, such as I/O and networking, file system access, and process isolation. Security bugs in the operating system can therefore undermine the security of everything running at higher layers of the software stack. Moreover, operating systems are ubiquitous and control everything from small embedded devices to supercomputers. Formal methods are important in ensuring their functional correctness as well as the absence of different classes of security problems.

5.4.1 Functional Correctness of Kernel Components

Verification of functional correctness is an essential part of verifying the security of kernel components. If parts of the kernel do not do their job properly, then it will be impossible to secure applications running on top. For example, it is essential that a multitasking kernel enforces *data separation* so that applications in one partition cannot read or modify the data in other partitions. There is also *temporal separation* where processes use resources sequentially and resources assigned to one application are properly sanitised before being assigned to another. Also important is *damage limitation* whereby the kernel limits the effect of compromises so that the compromise of one component does not impact others, or the operating system itself. These requirements are mandated by standards like the Common Criteria and ISO/IEC 15408 and are therefore demanded for certifying high-assurance systems like operating systems. Establishing them is an important part of functional verification.

We note in this regard that the distinction between functional requirements and non-functional requirements, which typically include security requirements (such as information flow control or absence of particular bug classes), is somewhat fuzzy. Functional requirements describe what the system should do. Generally they can be described via goal or I/O-oriented descriptions defining the system's purpose, or in terms of system states and traces. In contrast, non-functional requirements are usually seen as constraints, pertaining to aspects like usability, scalability, maintainability and testability. But the boundary is particularly blurred when it comes to security. As an example, memory safety is not the goal of an operating system. However, without memory safety the functions that the operating system should perform will not always work correctly, especially in the presence of an adversary who can find and exploit memory corruption vulnerabilities.

The seL4 project is the first, and probably best known, operating system verification effort, which formally verified a complete, general-purpose operating system kernel. The seL4 microkernel is a member of the L4 family, comprising 8,700 lines of C code and 600 lines of assembler. It was verified [184] from its abstract specification all the way down to its C implementation. Functional correctness for seL4 states that the implementation strictly

follows the abstract specification that precisely describes the kernel's behaviour. This includes standard safety properties such as requiring that the kernel will never crash or perform unsafe operations.

The verification of seL4 was carried out in Isabelle/HOL and uses refinement to establish a formal correspondence between three specifications of the system at different levels of abstraction. The first specification is an *abstract specification* that completely describes the kernel's behaviour by specifying the kernel's outer interface, e.g., system calls and their effects, as well as the effects of interrupts and system faults. The second specification is an *executable specification* written in a functional programming language and the third specification is seL4's actual *C implementation*. All three specifications are given in higher-order logic and for the third specification this necessitates also specifying a formal semantics for the subset of C used. At the heart of the proof are the refinement steps, showing that the executable specification refines the abstract specification and that the C specification refines the executable one. This guarantees that all Hoare logic properties established for the abstract specification also hold for the refined models, in particular the kernel's source code.

Verifying seL4 was a major effort, requiring roughly 20 person-years. This included ca. 9 person-years for developing formal language frameworks, proof tools, support for proof automation, and libraries. The seL4-specific proof took ca. 11 person-years. Subsequent work later extended the formalisation to also prove strong information flow control properties [221].

While seL4 is exemplary, it is by no means a lone example of what is possible with enough time and sweat. There have been dozens of formal methods projects that specify and verify substantial parts of operating systems and different variants thereof, e.g., hypervisors, separation kernels, partitioning kernels, security kernels, and microkernels [222]. Other projects that have proved the correctness of simple hypervisor-like OS kernels are CertiKOS [223] and ExpressOS [224]. In contrast, other projects focus on functional correctness for just selected critical parts of the kernel. For example, Jitk [225] is an infrastructure for building in-kernel interpreters using a just-in-time (JIT) compiler that translates high-level languages into native code like x86. Jitk uses CompCert, a verified compiler infrastructure [226] to guarantee that the translation is semantics-preserving. Other examples are FSCQ [227] and Cogent [228], which focus on the verification of file systems, and Microsoft's SLAM model checker [4], which uses predicate abstraction techniques to verify the correctness of Windows device drivers.

5.4.2 Absence of Bug Classes

Rather than verifying all or parts of an operating system (or any program, for that matter), formal methods may instead target eliminating specific kinds of security-critical vulnerabilities. Important examples, formulated positively in terms of desirable properties (which guarantee the absence of the vulnerabilities) are ensuring type and memory safety, or control flow integrity. We shall give examples of each of these below. These properties are non-negotiable prerequisites for software that needs any sort of integrity or reliability at all. One gets considerable benefits proving such properties, even if one is not aiming for full functional correctness.

Both type and memory safety guarantees are provided for the Verve operating system [229]. Type safety ensures that all operations are well typed; for example a stored Boolean value is not later retrieved as an integer. Memory safety, discussed in Section 5.2 ensures that the operations do not overflow assigned memory or deference dangling pointers. These properties rule out common security problems such as buffer overflows. All of Verge's

assembly language instructions (except those for the boot loader) are mechanically verified for safety; this includes device drivers, garbage collectors, schedulers, etc. To achieve this, Verve is written in a safe dialect of C# that is compiled to a typed assembly language (TAL) that runs on x86 hardware, and verification is based on a Hoare verifier (Boogie) that leverages the Z3 SMT solver for automation. Verve was later used as part of the Ironclad project [230] to build secure applications.

Control-Flow Integrity (CFI) was discussed in the Operating Systems & Virtualisation CyBOK Knowledge Area [189] and the Software Security CyBOK Knowledge Area [9]. CFI techniques are a form of dynamic analysis (cf. Section 2.3.4) where at runtime a program is monitored and regulated to prevent adversaries from altering a program's control flow to execute code of their choosing, e.g., for return-oriented programming. Efficient algorithms for dynamic analysis have made their way into modern compiler tool chains like GCC and LLVM [231]. Additionally, in the SVA [232] and KCoFI [233] projects, CFI and memory safety protection have been implemented into execution environments that can be used to protect commodity operating systems.

5.5 Web-based Applications

The Web & Mobile Security CyBOK Knowledge Area [234] introduces Web and mobile security. One key takeaway is that the Web ecosystem is tremendously complex. It comprises browsers, browser extensions, servers, protocols, and specialised languages, all of which interact in complex ways. Hence an important role of formal methods is to formalise these different parts of the Web and provide analysis techniques to help understand the properties of the individual components and their interaction. In light of the Web's complexity, various approaches have emerged to tackle different aspects of this problem; see [235] for a survey of the scope and applications of formal methods in this domain.

5.5.1 Web Programming

The JavaScript programming language is a core Web technology that runs in browsers and supports interactive Web pages. JavaScript is known to be rather quirky [236] and a source of security bugs. Given this complexity, an important application of formal methods is to provide JavaScript with a formal semantics. This was accomplished by [237], who formalised the JavaScript (ECMA) standard in the Coq proof assistant. Using this formalization, they verified reference interpreter in Coq, from which they extracted an executable OCaml interpreter. Moreover, by running the interpreter on existing JavaScript test suites, they were able to clarify and improve ambiguous parts of the standard.

Formal methods can also be used to provide support for developing JavaScript applications in a disciplined way. This idea is pursued in [238] who propose developing secure JavaScript applications in F* and using a verified compiler to produce JavaScript. As previously explained, F* is a dependently typed language with a clear semantics and verification support. The authors of [238] verify a full-abstraction result that allows programmers to reason about their programs with respect to F*'s semantics, rather than that of JavaScript. An alternative approach to improving the security of JavaScript is to enhance JavaScript interpreters to enforce information flow control and thereby ensure that sensitive data does not leave the browser [239].

Finally, one can use formal methods to explore alternative programming languages for the

Web. WebAssembly is a prominent example [240, 241]. This language features a formal semantics and there is support for verifying functional properties, information flow security, and timing side-channel resistance of WebAssembly programs.

5.5.2 Web Components

Formal methods can be used to analyse the properties of individual Web components and improve their security. We give two examples: Web browsers and Web protocols.

An executable, operational semantics of the core functionality of a web browser is presented in [242]. Their model accurate captures asynchronous execution within the browser, browser windows, DOM trees, cookies, HTTP requests and responses, and a scripting language with first-class functions, dynamic evaluation, and AJAX requests. Such a model provides a valuable basis for formalizing and experimenting with different security policies and mechanisms, such as solutions to web-script security problems like cross-site scripting.

One may also focus on the security of the underlying protocols. For example, [243] modelled a Web-based single sign-on protocol given by the OASIS Security Assertion Markup Language (SAML) 2.0 Web Browser SSO Profile. This protocol is used by identity providers such as Google for single sign-on, on the Web. However, analysis of the protocol's authenticity properties using the SATMC model checker revealed severe flaws that allowed a dishonest service provider to impersonate users [243]. Here, as is often the case when using formal methods, the attack suggested a fix, which was subsequently verified.

5.5.3 Component Interaction

In contrast to focusing on individual Web components, one may consider the interaction of the Web's components by modelling subsets of the Web ecosystem, at an appropriate level of abstraction. [244] create a model of key Web components and their interactions in Alloy, a modelling tool based on first-order logic. The components include browsers, servers, scripts, HTTP, and DNS. Additionally they model different adversary capabilities like Web attackers who can create malicious web-sites or active network attackers. Finally, they specify different invariants and properties that are typically required for secure browsing; for example, when a server receives an HTTP request, it often wishes to ensure that the request was generated by a trusted principal and not an attacker.

Specifications like the above bring us into the standard setting of formal methods for security: given a model of the system and an attacker, one wishes to verify that all behaviours satisfy the given property. [244] uses Alloy's analyser to answer this and thereby find bugs in existing security mechanisms and analyse proposed improvements to the Web. For example, they analyze and find attacks on: a proposed cross-site request forgery defense based on the Origin header, Cross-Origin Resource Sharing; a proposal to use Referrer validation to prevent cross-site scripting; a new functionality in the HTML5 form element; and WebAuth, a Kerberos-based single sign-on system. An analogous approach is taken by [245], who provide a detailed model of core security aspects of the Web infrastructure, staying as close as possible to the underlying Web standards. The model was used to provide a rigourous security analysis of the BrowserID system.

5.6 Full-stack Verification

We have considered various options for using formal methods to improve security across the system stack. Ideally, the entire stack, or as much of it as is possible, comes with guarantees that are linked across levels.

The first such stack was specified in pure Lisp and verified with the Boyer-Moore (NQTHM) prover, a predecessor of the ACL2 theorem prover. The focus was on the hardware and operating system layers [185] and establishing functional correctness, not security per se. The components verified included a code generator for a high-level language, an assembler and linking loader, a simple operating system, and a microprocessor. These layers were connected by downward and upward functions, mapping down and up the system stack. The downward functions formalised that: the compiler maps high-level language states to assembly language states; an assembler, linker and loader that, when composed, map the assembly language states to states of their machine architecture; and a function that maps architecture states to register transfer level (RTL) states. The upward maps formalised the high-level data represented by a region of a low-level state. Theorems were proved in NQTHM that related these layers. Finally, pre-conditions and post-conditions for the applications programs were defined and the programs were proved correct with respect to the high-level language semantics. All these parts were put together in a "warranty theorem" that informally states the following [185]:

> "Suppose one has a high-level language state satisfying the pre-condition of a given application program [...]. Construct the RTL state obtained from the high-level state by successively compiling, assembling, linking, and loading. Run the RTL machine on the constructed state. Map certain data regions up, using the partial inverse functions, producing some high-level data objects. Then the postcondition holds of that high-level data."

This theorem was formalised and proved using the NQTHM theorem prover, thereby linking all the layers. This stack, albeit somewhat simplistic, captures the essence of full-stack verification.

The vision of full-stack verification is being pursued on a larger scale within the DeepSpec [246] project. This project has the goal of developing fully-verified software and hardware, including operating system kernels [223], cryptographic primitives and protocols [247], a message system [248], and a networked server [249]. Verification uses the Verified Software Toolchain (VST) [250], which is a framework for verifying C programs via a separation logic embedded in Coq.

6 CONFIGURATION

[251, 252, 253]

The security of a system depends not only on its implementation, but also its configuration and operation. When configuring a system, accounts must be created, credentials and certificates must be distributed or configured, and authorisation policies must be specified for numerous systems at different levels of the stack, e.g., the operating system, middleware, applications, and network devices. Clearly this must also be done correctly and formal methods can help to determine whether this is the case, or even assist with the configuration process, for example, by compiling high-level policies to low-level configurations.

We will expand on this in the following subsections, with a primary focus on the correctness (with respect to high-level *properties*) of the authorisations (also called the *policy*) configured to enforce access control. This problem is important as access control mechanisms are ubiquitous, the specification and management of access control policies is challenging, and misconfiguration is the source of many security problems in practice. See the Authentication, Authorisation & Accountability CyBOK Knowledge Area [254] for more on the challenges in this regard, as well as for a discussion on logics for access control and delegation.

6.1 Policy Analysis

As observed in the Authentication, Authorisation & Accountability CyBOK Knowledge Area [254], there is a wide spectrum of policy languages in which administrators can specify access control policies. Many of these languages are low-level and not particularly user friendly. Moreover, organisations and their IT setup can be complex, which is reflected in their access control policies. A large-scale organisation may have millions of users and access may need to be regulated across numerous and diverse resources including applications, files and data sets. Even when industry-standard policy languages are used, such as for role-based access control (RBAC) or attribute-based access control (ABAC), the resulting policies are voluminous and detailed. Take an RBAC setup in a typical organisation: there may be hundreds of roles and thousands of rules and it is extremely challenging to get this right, especially as authorisations evolve over time. The problem here is analogous to program verification: how does one know whether what is written is really what is wanted?

Early research on formalising access control, like the Harrison, Ruzzo, Ullman (HRU) model described in the Authentication, Authorisation & Accountability CyBOK Knowledge Area [254], investigated notions of *safety* within the context of access control matrices and updates to them. The authors of this seminal work posed the problem of *safety analysis*, asking whether access rights can be leaked in undesirable ways. Since then, researchers have looked into both:

1. handling different, more realistic policy languages than those based on the access-control matrices used in HRU, and
2. reasoning about more general properties (safety is interesting, albeit rather specialised).

One generalisation of safety analysis is *security analysis* [255], where the property concerns whether an access control system preserves invariants that encode desired security properties, i.e., whether these properties are preserved across state changes. [256] explores this problem in the context of role-based access control. The authors study the complexity of this problem (showing it is in PSPACE) and provide algorithmic solutions based, for example, on model checking.

In [251], the authors show how to generalise both (1) the policy language and (2) the properties, simultaneously. They show that many policy languages, including various RBAC extensions and ABAC, can be directly expressed in a fragment of first-order logic, which the authors identify and call FORBAC. Moreover, the properties that one expects configurations to satisfy can also be formalised within FORBAC. Hence one can use first-order deduction to analyse whether configurations satisfy their desired properties. For FORBAC, [251] shows that SMT solvers suffice, that this approach scales in practice, and it can be used to verify or find errors in industrial-scale access control configurations.

A related problem to policy analysis is understanding policy changes, given that policies are

maintained and evolve over time. This problem is known as *change-impact analysis*. [252] presents Margrave, which is such an analysis tool for access-control policies written in the XACML standard and another version of Margrave handles firewall policy languages [257]. Like the tools mentioned above, given an XACML policy and a property, Margrave can determine whether the policy satisfies the property. In addition, for change-impact analysis, it can take two policies and summarise the semantic differences between them — what is allowed in one but not the other — by computing a compact representation in the form of a multi-terminal Binary Decision Diagram.

Another approach to finding errors in policies, in the context of network firewalls, is taken by [258]. Firewall rules are generally written in low level, platform-specific languages, their control flow may be non-trivial, and they are further complicated by Network Address Translation (NAT). Unsurprisingly, it is easy for administrators to make mistakes when specifying configurations. The idea behind [258] is to reverse the compilation procedure and use *decompilation* to obtain a more high-level description of a given firewall configuration. Specifically, their tool, called FireWall Synthesizer, translates configurations from different popular firewall systems into a common intermediate language; from configurations in the intermediate language, the Z3 SMT solver is used to synthesise an abstract firewall policy, in tabular form. The generated specification plays the role of a property, in that it is much easier for system administrators to understand than the original policy. The tool also supports the comparison of different configuration versions.

6.2 Specification-based Synthesis

Rather than having administrators directly configure policies and afterwards, if at all, prove that they satisfy desired properties, an alternative is to specify the properties and synthesise policy configurations from them. The advantage of this approach is that if the property language is sufficiently high-level, then it is presumably easier to express correctly the property than the policy that implements it. This is analogous to the advantages of using high-level programming languages over assembler, or the synthesis of controllers from declarative specifications of their properties.

One example of this is the tool [253] that translates policies written in a high-level policy language to policies in XACML, which is a low-level, machine-oriented, industrial standard policy language. The high-level language used is a first-order policy specification language that allows authorisation rules to be defined based on arbitrary conditions. The translation to XACML combines syntax-driven translation with query answering. It exemplifies a common translation-oriented approach of using simple deductive reasoning to expand a policy written succinctly in a high-level language to a more verbose one written in a lower-level (but, equally expressive) language.

A second, rather different example is found in [259] and [260], both of which provide methods to construct runtime monitors for workflow enforcement engines that enforce access control policies involving separation of duty, which is the security principle that requires different users to carry out different tasks in a work flow. For example, the user who is responsible for financial transactions in a bank should be different from the user who audits the transactions. Roughly speaking, both approaches require that one specifies which pairs (or sets) of tasks in a workflow must be handled by different users, along with other access-control constraints. The problem is then to generate a runtime monitor that enforces all these constraints, while being as liberal as possible so that the workflow can successfully complete. [259] solves

the problem using techniques based on symbolic model checking. In contrast, [260] tackles the problem by providing a specification language for access-control and separation of duty constraints based on the process algebra CSP. Namely, given a CSP specification ϕ, an execution monitor (cf. Authentication, Authorisation & Accountability CyBOK Knowledge Area [254]) is synthesised corresponding to the transition system for ϕ defined by CSP's operational semantics.

CROSS-REFERENCE OF TOPICS VS REFERENCE MATERIAL

Topics	Cites
1 Motivation	
2 Foundations, Methods and Tools	[14, 19, 20, 21, 22, 23]
3 Hardware	[75, 76, 77, 78]
4 Cryptographic Protocols	[95, 96, 97, 98, 99, 100]
5 Software and Large-Scale Systems	[179, 180, 181, 182, 183, 184, 185]
6 Configuration	[251, 252, 253]

ACKNOWLEDGEMENTS

The author would like to specially thank Christoph Sprenger and Kenny Paterson for their valuable input.

REFERENCES

[1] W. Conradie and V. Goranko, *Logic and Discrete Mathematics - A Concise Introduction*. Wiley, 2015. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118751272.html

[2] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[3] T. Nipkow and G. Klein, *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[4] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside microsoft," in *Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–20.

[5] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook, "Continuous formal verification of Amazon s2n," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 430–446.

[6] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services uses formal methods," *Commun. ACM*, vol. 58, no. 4, p. 66–73, Mar. 2015. [Online]. Available: https://doi.org/10.1145/2699417

[7] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[8] L. Williams, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Secure Software Lifecycle, version 1.0.2. [Online]. Available: https://www.cybok.org/
[9] F. Piessens, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Software Security, version 1.0.1. [Online]. Available: https://www.cybok.org/
[10] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, p. 92–106, Dec. 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895
[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, Feb. 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374
[12] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.
[13] G. Stringhini, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Adversarial Behaviours, version 1.0.1. [Online]. Available: https://www.cybok.org/
[14] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, p. 1157–1210, Sep. 2010.
[15] F. Schneider, "Blueprint for a science of cybersecurity," *The Next Wave*, vol. 19, no. 2, pp. 47–57, 2012.
[16] C. Herley and P. C. Van Oorschot, "SoK: Science, security and the elusive goal of security as a scientific pursuit," in *2017 IEEE Symposium on Security and Privacy*, 2017, pp. 99–120.
[17] D. Basin and S. Capkun, "The research value of publishing attacks," *Commun. ACM*, vol. 55, no. 11, pp. 22–24, November 2012. [Online]. Available: http://doi.acm.org/10.1145/2366316.2366324
[18] M. T. Dashti and D. A. Basin, "Security testing beyond functional tests," in *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, 2016, pp. 1–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30806-7_1
[19] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distrib. Comput.*, vol. 2, no. 3, p. 117–126, Sep. 1987. [Online]. Available: https://doi.org/10.1007/BF01782772
[20] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.
[21] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Cham: Springer International Publishing, 2018, pp. 305–343. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8-11
[22] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973
[23] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832608000775
[24] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.
[25] D. E. Bell and L. J. L. Padula, "Secure computer system: Unified exposition and multics interpretation," 1976.
[26] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5,

p. 236–243, May 1976. [Online]. Available: https://doi.org/10.1145/360051.360056
[27] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security - Tools for Analysis and Verification*, ser. NATO Science for Peace and Security Series - D: Information and Communication Security, T. Nipkow, O. Grumberg, and B. Hauptmann, Eds. IOS Press, 2012, vol. 33, pp. 319–347. [Online]. Available: https://doi.org/10.3233/978-1-61499-028-4-319
[28] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," *CoRR*, vol. abs/1401.4492, 2014. [Online]. Available: http://arxiv.org/abs/1401.4492
[29] B. Finkbeiner, M. N. Rabe, and C. Sánchez, "Algorithms for model checking HyperLTL and HyperCTL*," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 30–48.
[30] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, 2003, pp. 29–43.
[31] D. McCullough, "Noninterference and the composability of security properties," in *2012 IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, apr 1988, p. 177. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SECPRI.1988.8110
[32] J. McLean, "A general theory of composition for a class of "possibilistic" properties," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 53–67, 1996.
[33] D. Park, "Concurrency and automata on infinite sequences," in *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. Berlin, Heidelberg: Springer-Verlag, 1981, p. 167–183.
[34] R. Milner, *Communication and Concurrency*. USA: Prentice-Hall, Inc., 1989.
[35] R. Gorrieri and C. Versari, *Introduction to Concurrency Theory: Transition Systems and CCS*, 1st ed. Springer Publishing Company, Incorporated, 2015.
[36] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The spi calculus," in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, ser. CCS '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 36–47. [Online]. Available: https://doi.org/10.1145/266420.266432
[37] ——, "A bisimulation method for cryptographic protocols," *Nordic J. of Computing*, vol. 5, no. 4, p. 267–303, Dec. 1998.
[38] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," *SIGPLAN Not.*, vol. 36, no. 3, p. 104–115, Jan. 2001. [Online]. Available: https://doi.org/10.1145/373243.360213
[39] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Found. Trends Priv. Secur.*, vol. 1, no. 1–2, p. 1–135, Oct. 2016. [Online]. Available: https://doi.org/10.1561/3300000004
[40] B. Blanchet, "Automatic proof of strong secrecy for security protocols," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 2004, pp. 86–100.
[41] V. Cheval, "Automatic verification of cryptographic protocols: privacy-type properties," PhD Thesis, Laboratoire Spécification et Vérification, ENS Cachan, France, Dec. 2012.
[42] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 117–148, 2003. [Online]. Available: https://doi.org/10.1016/S0065-2458(03)58003-2
[43] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on Common Lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, 1997.

[44] J. S. Moore, "Milestones from the pure lisp theorem prover to ACL2," *Formal Aspects Comput.*, vol. 31, no. 6, pp. 699–732, 2019. [Online]. Available: https://doi.org/10.1007/s00165-019-00490-3

[45] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283. [Online]. Available: https://doi.org/10.1007/3-540-45949-9

[46] J. Harrison, "HOL Light: An overview," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 60–66.

[47] Y. Bertot and P. Castran, *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[48] G. Huet, G. Kahn, and C. Paulin-Mohring, *The* Coq *Proof Assistant - A tutorial - Version 7.1*, Oct. 2001, http://coq.inria.fr.

[49] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[50] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 530–535. [Online]. Available: https://doi.org/10.1145/378239.379017

[51] J. P. Marques Silva and K. A. Sakallah, *Grasp—A New Search Algorithm for Satisfiability*. Boston, MA: Springer US, 2003, pp. 73–89. [Online]. Available: https://doi.org/10.1007/978-1-4615-0292-0_7

[52] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[53] F. Corzilius, U. Loup, S. Junges, and E. Ábrahám, "SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox," in *Theory and Applications of Satisfiability Testing – SAT 2012*, A. Cimatti and R. Sebastiani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 442–448.

[54] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, p. 69–77, Sep. 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394

[55] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[56] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177.

[57] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[58] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 2000.

[59] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[60] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, p. 293–318, Sep. 1992. [Online]. Available: https://doi.org/10.1145/136035.136043

[61] D. Peled, "Ten years of partial order reduction," in *Computer Aided Verification*, A. J. Hu and M. Y. Vardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 17–28.

[62] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, p. 279–295, May 1997. [Online]. Available: https://doi.org/10.1109/32.588521
[63] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364.
[64] P. Gardiner, M. Goldsmith, J. Hulance, D. Jackson, A. Roscoe, B. Scattergood, and P. Armstrong, "FDR2 user manual," 2000.
[65] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591.
[66] W. Lee, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Malware & Attack Technology, version 1.0.1. [Online]. Available: https://www.cybok.org/
[67] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook, "Continuous formal verification of Amazon s2n," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 430–446.
[68] B. Cook, "Formal reasoning about the security of Amazon Web Services," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 38–47.
[69] U. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, 2000, pp. 246–255.
[70] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup, "Efficient monitoring of hyperproperties using prefix trees," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 6, pp. 729–740, 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00552-5
[71] K. Havelund and G. Roşu, "An overview of the runtime verification tool Java PathExplorer," *Formal Methods in System Design*, vol. 24, no. 2, pp. 189–215, Mar 2004.
[72] D. Basin, F. Klaedtke, and S. Müller, "Monitoring security policies with metric first-order temporal logic," in *15th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM Press, 2010, pp. 23–33.
[73] E. Arfelt, D. Basin, and S. Debois, "Monitoring the GDPR," in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Cham: Springer International Publishing, 2019, pp. 681–699.
[74] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu, "Monitoring metric first-order temporal properties," *J. ACM*, vol. 62, no. 2, pp. 15:1–15:45, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2699444
[75] R. Alur, T. A. Henzinger, and M. Y. Vardi, "Theory in practice for system design and verification," *ACM SIGLOG News*, vol. 2, no. 1, p. 46–51, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2728816.2728827
[76] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Advances in Cryptology - EUROCRYPT 2009*, A. Joux, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 443–461.
[77] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2756550
[78] M. Bond and R. Anderson, "API-level attacks on embedded systems," *Computer*, vol. 34, no. 10, pp. 67–75, 2001.
[79] I. Verbauwhede, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Hardware Security, version 1.0.1. [Online]. Available: https://www.cybok.org/

[80] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, p. 123–193, Apr. 1999. [Online]. Available: https://doi.org/10.1145/307988.307989

[81] T. Kropf, *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 1999.

[82] E. Clarke and D. Kroening, "Hardware verification using ANSI-C programs as a reference," in *Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003.*, 2003, pp. 308–311.

[83] J. Sawada and W. A. Hunt Jr., "Verification of FM9801: an out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability," *Formal Methods Syst. Des.*, vol. 20, no. 2, pp. 187–222, 2002. [Online]. Available: https://doi.org/10.1023/A:1014122630277

[84] M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin, *Formal Verification of Partition Management for the AAMP7G Microprocessor*. Boston, MA: Springer US, 2010, pp. 175–191. [Online]. Available: https://doi.org/10.1007/978-1-4419-1539-9_6

[85] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, 2005, pp. 255–269.

[86] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 398–412.

[87] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Comput. Secur.*, vol. 8, no. 2,3, p. 141–158, Aug. 2000.

[88] S. Micali and L. Reyzin, "Physically observable cryptography," in *Theory of Cryptography*, M. Naor, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 278–296.

[89] B. Köpf and D. A. Basin, "Automatically deriving information-theoretic bounds for adaptive side-channel attacks," *Journal of Computer Security*, vol. 19, no. 1, pp. 1–31, 2011.

[90] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: A DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 174–189. [Online]. Available: https://doi.org/10.1145/3314221.3314605

[91] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 15–26. [Online]. Available: https://doi.org/10.1145/3213846.3213851

[92] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 328–343.

[93] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *SIGPLAN Not.*, vol. 50, no. 4, p. 503–516, Mar. 2015. [Online]. Available: https://doi.org/10.1145/2775054.2694372

[94] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, "Attacking and fixing PKCS#11 security tokens," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 260–269. [Online]. Available: https://doi.org/10.1145/1866307.1866337

[95] B. Blanchet and D. Pointcheval, "Automated security proofs with sequences of games," in *CRYPTO'06*, ser. Lecture Notes in Computer Science, C. Dwork, Ed., vol. 4117. Santa Barbara, CA: Springer, Aug. 2006, pp. 537–554.

[96] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, 2001, pp. 82–96.

[97] L. Paulson, "The inductive approach to verifying cryptographic protocols," *J. Computer Security*, vol. 6, pp. 85–128, 1998. [Online]. Available: http://www.cl.cam.ac.uk/users/lcp/papers/Auth/jcs.pdf

[98] G. Lowe, "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 147–166.

[99] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A. Roscoe, *The Modelling and Analysis of Security Protocols: The CSP Approach*, 1st ed. Addison-Wesley Professional, 2000.

[100] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, june 2012, pp. 78 –94.

[101] D. Basin, C. Cremers, and C. Meadows, *Model Checking Security Protocols*. Springer, 2018, ch. 24, pp. 727–762.

[102] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," Cryptology ePrint Archive, Report 2019/1393, 2019, https://eprint.iacr.org/2019/1393.

[103] D. A. Basin, C. J. F. Cremers, K. Miyazaki, S. Radomirovic, and D. Watanabe, "Improving the security of cryptographic protocol standards," *IEEE Security & Privacy*, vol. 13, no. 3, pp. 24–31, 2015. [Online]. Available: http://dx.doi.org/10.1109/MSP.2013.162

[104] D. Basin, C. Cremers, and S. Meier, "Provably repairing the ISO/IEC 9798 standard for entity authentication," in *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, ser. Lecture Notes in Computer Science, P. Degano and J. D. Guttman, Eds., vol. 710.1007/978-3-319-40667-1_185. Springer, 2012, pp. 129–148.

[105] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. R. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoue, "Everest: Towards a verified, drop-in replacement of HTTPS," in *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, ser. LIPIcs, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 1:1–1:12. [Online]. Available: https://doi.org/10.4230/LIPIcs.SNAPL.2017.1

[106] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552.

[107] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1773–1788. [Online]. Available: https://doi.org/10.1145/3133956.3134063

[108] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5G authentication," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA:

Association for Computing Machinery, 2018, p. 1383–1396. [Online]. Available: https://doi.org/10.1145/3243734.3243846

[109] D. A. Basin, R. Sasse, and J. Toro-Pozo, "The EMV standard: Break, Fix, Verify," in *42nd IEEE Symposium on Security and Privacy (Oakland S&P)*, 2021, to appear.

[110] ——, "Card brand mixup attack: Bypassing the PIN in non-Visa cards by using them for Visa transactions," in *30th USENIX Security Symposium*, 2021, to appear.

[111] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Information Theory*, vol. 29, no. 2, pp. 198–207, 1983. [Online]. Available: https://doi.org/10.1109/TIT.1983.1056650

[112] G. Lowe, "A hierarchy of authentication specification," in *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society, 1997, pp. 31–44. [Online]. Available: https://doi.org/10.1109/CSFW.1997.596782

[113] G. Bella, F. Massacci, and L. C. Paulson, "Verifying the SET registration protocols," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 77–87, 2003.

[114] S. Meier, C. Cremers, and D. Basin, "Strong invariants for the efficient construction of machine-checked protocol security proofs," in *23rd IEEE Computer Security Foundations Symposium*. Los Alamitos, USA: IEEE Computer Society, 7 2010, pp. 231–245.

[115] S. Meier, C. Cremers, and D. A. Basin, "Efficient construction of machine-checked symbolic protocol security proofs," *Journal of Computer Security*, vol. 21, no. 1, pp. 41–87, 2013.

[116] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90224-P

[117] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[118] J. Lallemand, D. A. Basin, and C. Sprenger, "Refining authenticated key agreement with strong adversaries," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017, pp. 92–107.

[119] C. Sprenger and D. A. Basin, "Refining security protocols," *Journal of Computer Security*, vol. 26, no. 1, pp. 71–120, 2018. [Online]. Available: https://doi.org/10.3233/JCS-16814

[120] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "Multiset rewriting and the complexity of bounded security protocols," *Journal of Computer Security*, vol. 12, no. 2, pp. 247–311, Apr. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1017273.1017276

[121] J. K. Millen, S. C. Clark, and S. B. Freedman, "The Interrogator: Protocol security analysis," *IEEE Trans. Software Eng.*, vol. 13, no. 2, pp. 274–288, 1987.

[122] D. Longley and S. Rigby, "An automatic search for security flaws in key management schemes," *Computers & Security*, vol. 11, no. 1, pp. 75–89, 1992.

[123] C. Meadows, "The NRL Protocol Analyzer: An overview," *J. Log. Program.*, vol. 26, no. 2, pp. 113–131, 1996.

[124] G. Lowe, "Casper: A compiler for the analysis of security protocols," *J. Comput. Secur.*, vol. 6, no. 1–2, p. 53–84, Jan. 1998.

[125] A. Armando, R. Carbone, and L. Compagna, "SATMC: A SAT-based model checker for security protocols, business processes, and security apis," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 2, p. 187–204, Apr. 2016. [Online]. Available: https://doi.org/10.1007/s10009-015-0385-y

[126] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P.-C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA tool for the automated

validation of internet security protocols," in *Proceedings of CAV'2005*, ser. LNCS 3576. Springer-Verlag, 2005, pp. 281–285.

[127] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *25th International Conference on Computer Aided Verification (CAV 2013)*, ser. LNCS, N. Sharygina and H. Veith, Eds., vol. 8044. Saint Petersburg, Russia: Springer, July 2013, pp. 696–701.

[128] B. Blanchet, "Automatic verification of correspondences for security protocols," *J. Comput. Secur.*, vol. 17, no. 4, p. 363–434, Dec. 2009.

[129] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," in *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, 2005, pp. 331–340.

[130] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006, pp. 14 pp.–152.

[131] D. Basin and C. Cremers, "Know your enemy: Compromising adversaries in protocol analysis," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 2, pp. 7:1–7:31, Nov. 2014. [Online]. Available: http://doi.acm.org/10.1145/2658996

[132] M. D. Ryan and B. Smyth, "Applied pi calculus," in *Formal Models and Techniques for Analyzing Security Protocols*, V. Cortier and S. Kremer, Eds. IOS Press, 2011, ch. 6.

[133] V. Cortier and S. Delaune, "A method for proving observational equivalence," in *2009 22nd IEEE Computer Security Foundations Symposium*, 2009, pp. 266–276.

[134] D. Basin, J. Dreier, and R. Sasse, "Automated symbolic proofs of observational equivalence," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1144–1155. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813662

[135] S. Santiago, S. Escobar, C. Meadows, and J. Meseguer, "A formal definition of protocol indistinguishability and its verification using maude-npa," in *Security and Trust Management*, S. Mauw and C. D. Jensen, Eds. Cham: Springer International Publishing, 2014, pp. 162–177.

[136] M. Baudet, V. Cortier, and S. Delaune, "YAPA: A generic tool for computing intruder knowledge," *ACM Trans. Comput. Logic*, vol. 14, no. 1, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2422085.2422089

[137] c. Ciobâcă, S. Delaune, and S. Kremer, "Computing knowledge in security protocols under convergent equational theories," in *Proceedings of the 22nd International Conference on Automated Deduction*, ser. CADE-22. Berlin, Heidelberg: Springer-Verlag, 2009, p. 355–370. [Online]. Available: https://doi.org/10.1007/978-3-642-02959-2_27

[138] B. Conchinha, D. Basin, and C. Caleiro, "Efficient decision procedures for message deducibility and static equivalence," in *Formal Aspects of Security and Trust*, P. Degano, S. Etalle, and J. Guttman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 34–49.

[139] M. Baudet, "Deciding security of protocols against off-line guessing attacks," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 16–25. [Online]. Available: https://doi.org/10.1145/1102120.1102125

[140] R. Corin, J. Doumen, and S. Etalle, "Analysing password protocol security against off-line dictionary attacks," *Electronic Notes in Theoretical Computer Science*, vol. 121, pp. 47–63, 2005, proceedings of the 2nd International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066105000241

[141] M. Stoelinga, "An introduction to probabilistic automata," *Bull. EATCS*, vol. 78, pp. 176–198, 2002.

[142] C. Baier, L. de Alfaro, V. Forejt, and M. Kwiatkowska, *Model Checking Probabilistic Systems*. Cham: Springer International Publishing, 2018, pp. 963–999. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8-28

[143] M. Kwiatkowska, G. Norman, and D. Parker, "Quantitative analysis with the probabilistic model checker PRISM," *Electron. Notes Theor. Comput. Sci.*, vol. 153, no. 2, p. 5–31, May 2006. [Online]. Available: https://doi.org/10.1016/j.entcs.2005.10.030

[144] S. Basagiannis, P. Katsaros, A. Pombortsis, and N. Alexiou, "Probabilistic model checking for the quantification of DoS security threats," *Comput. Secur.*, vol. 28, no. 6, p. 450–465, Sep. 2009. [Online]. Available: https://doi.org/10.1016/j.cose.2009.01.002

[145] V. Shmatikov, "Probabilistic analysis of an anonymity system," *J. Comput. Secur.*, vol. 12, no. 3,4, p. 355–377, May 2004.

[146] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for web transactions," *ACM Trans. Inf. Syst. Secur.*, vol. 1, no. 1, p. 66–92, Nov. 1998. [Online]. Available: https://doi.org/10.1145/290163.290168

[147] R. Lanotte, A. Maggiolo-Schettini, and A. Troina, "Automatic analysis of a non-repudiation protocol," in *Proc. 2nd International Workshop on Quantitative Aspects of Programming Languages (QAPL'04)*, 2004.

[148] G. Norman and V. Shmatikov, "Analysis of probabilistic contract signing," in *Formal Aspects of Security*, A. E. Abdallah, P. Ryan, and S. Schneider, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 81–96.

[149] V. Galpin, "Formal modelling of software defined networking," in *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. A. Furia and K. Winter, Eds., vol. 11023. Springer, 2018, pp. 172–193. [Online]. Available: https://doi.org/10.1007/978-3-319-98938-9_11

[150] R. Dimitrova, B. Finkbeiner, and H. Torfah, "Probabilistic hyperproperties of Markov decision processes," in *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. V. Hung and O. Sokolsky, Eds., vol. 12302. Springer, 2020, pp. 484–500. [Online]. Available: https://doi.org/10.1007/978-3-030-59152-6_27

[151] E. Ábrahám, E. Bartocci, B. Bonakdarpour, and O. Dobe, "Probabilistic hyperproperties with nondeterminism," in *Automated Technology for Verification and Analysis*, D. V. Hung and O. Sokolsky, Eds. Cham: Springer International Publishing, 2020, pp. 518–534.

[152] N. Smart, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Cryptography, version 1.0.1. [Online]. Available: https://www.cybok.org/

[153] N. Koblitz and A. J. Menezes, "Another look at "provable security"," *J. Cryptol.*, vol. 20, no. 1, p. 3–37, Jan. 2007. [Online]. Available: https://doi.org/10.1007/s00145-005-0432-z

[154] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *EUROCRYPT 2006*, ser. LNCS, vol. 4004. Springer, 2006, pp. 409–426.

[155] V. Shoup, "Sequences of games: A tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, 2004. [Online]. Available: http://eprint.iacr.org/2004/332

[156] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*. IEEE, 2001, pp. 136–145.

[157] U. Maurer, "Constructive cryptography – a new paradigm for security definitions and proofs," in *Joint Workshop on Theory of Security and Applications*. Springer, 2011, pp. 33–56.

[158] S. Halevi, "A plausible approach to computer-aided cryptographic proofs," 2005, shaih@alum.mit.edu 12949 received 15 Jun 2005. [Online]. Available: http://eprint.iacr.org/2005/181

[159] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *J. Cryptol.*, vol. 20, no. 3, p. 395, Jul. 2007.

[160] D. Micciancio and B. Warinschi, "Soundness of formal encryption in the presence of active adversaries," in *Theory of Cryptography*, M. Naor, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 133–151.

[161] M. Backes, D. Hofheinz, and D. Unruh, "CoSP: A general framework for computational soundness proofs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 66–78. [Online]. Available: https://doi.org/10.1145/1653662.1653672

[162] V. Cortier and B. Warinschi, "Computationally sound, automated proofs for security protocols," in *Programming Languages and Systems*, M. Sagiv, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 157–171.

[163] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *J. Autom. Reason.*, vol. 46, no. 3-4, pp. 225–259, 2011. [Online]. Available: https://doi.org/10.1007/s10817-010-9187-9

[164] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 04, pp. 193–207, oct 2008.

[165] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 90–101. [Online]. Available: https://doi.org/10.1145/1480881.1480894

[166] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *POST 2015*, ser. LNCS, vol. 9036. Springer, 2015, pp. 53–72.

[167] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," *Journal of Cryptology*, pp. 1–73, 2020.

[168] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Proceedings of the 31st Annual Conference on Advances in Cryptology*, ser. CRYPTO'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 71–90.

[169] Y. Lindell, *How to Simulate It – A Tutorial on the Simulation Proof Technique*. Cham: Springer International Publishing, 2017, pp. 277–346. [Online]. Available: https://doi.org/10.1007/978-3-319-57048-8_6

[170] A. Datta, R. Küsters, J. C. Mitchell, and A. Ramanathan, "On the relationships between notions of simulation-based security," in *Proceedings of the Second International Conference on Theory of Cryptography*, ser. TCC'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 476–494. [Online]. Available: https://doi.org/10.1007/978-3-540-30576-7_26

[171] M. Backes, B. Pfitzmann, and M. Waidner, "A universally composable cryptographic library," Cryptology ePrint Archive, Report 2003/015, 2003, https://eprint.iacr.org/2003/015.

[172] R. Kusters, "Simulation-based security with inexhaustible interactive turing machines," in *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, ser. CSFW '06. USA: IEEE Computer Society, 2006, p. 309–320. [Online]. Available:

https://doi.org/10.1109/CSFW.2006.30
[173] A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague, "Probabilistic bisimulation and equivalence for security analysis of network protocols," in *Foundations of Software Science and Computation Structures*, I. Walukiewicz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 468–483.
[174] S. Delaune, S. Kremer, and O. Pereira, "Simulation based security in the applied pi calculus," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Kannan and K. N. Kumar, Eds., vol. 4. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 169–180. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2009/2316
[175] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner, "Cryptographically sound theorem proving," in *19th IEEE Computer Security Foundations Workshop, Venice, Italy*. IEEE Computer Society, July 2006, pp. 153–166.
[176] A. Lochbihler, S. R. Sefidgar, D. Basin, and U. Maurer, "Formalizing constructive cryptography using CryptHOL," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, June 2019, pp. 152–15 214.
[177] D. Butler, D. Aspinall, and A. Gascón, "How to simulate it in Isabelle: Towards formal proof for secure multi-party computation," in *Interactive Theorem Proving*, M. Ayala-Rincón and C. A. Muñoz, Eds. Cham: Springer International Publishing, 2017, pp. 114–130.
[178] R. Canetti, A. Stoughton, and M. Varia, "EasyUC: Using EasyCrypt to mechanize proofs of universally composable security," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 167–16 716.
[179] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, p. 5–19, Sep. 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121
[180] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, p. 410–442, Oct. 2000. [Online]. Available: https://doi.org/10.1145/363516.363526
[181] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3324783
[182] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1789–1806. [Online]. Available: https://doi.org/10.1145/3133956.3134043
[183] A. Chlipala, "Mostly-automated verification of low-level programs in computational separation logic," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 234–245. [Online]. Available: https://doi.org/10.1145/1993498.1993526
[184] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Symposium on Operating Systems Principles (SOSP)*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220.
[185] W. R. Bevier, W. A. H. Jr., J. S. Moore, and W. D. Young, "An approach to systems verification," *J. Autom. Reason.*, vol. 5, no. 4, pp. 411–428, 1989. [Online]. Available: https://doi.org/10.1007/BF00243131
[186] H. Garavel and S. Graf, *Formal Methods for Safe and Secure Computers Systems - BSI*

Study 875. BSI German Federal Office for Information Security, 2013.

[187] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[188] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich, "Report on the NSF workshop on formal methods for security," NSF, USA, Tech. Rep., 2016.

[189] H. Bos, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Operating Systems & Virtualisation, version 1.0.1. [Online]. Available: https://www.cybok.org/

[190] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, p. 504–513, Jul. 1977. [Online]. Available: https://doi.org/10.1145/359636.359712

[191] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, no. 2–3, p. 167–187, Jan. 1996.

[192] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA: Springer US, 2007, pp. 291–307.

[193] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 129–142. [Online]. Available: https://doi.org/10.1145/268998.266669

[194] V. Simonet, "Flow Caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*, G. Hutton, Ed., Nottingham, United Kingdom, Mar. 2003, pp. 152–165.

[195] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 117–158, Jan. 2003, ©ACM.

[196] R. Chapman and A. Hilton, "Enforcing security and safety models with an information flow analysis tool," in *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time and Distributed Systems Using Ada and Related Technologies*, ser. SIGAda '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 39–46. [Online]. Available: https://doi.org/10.1145/1032297.1032305

[197] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, p. 37–61, Jan. 1985. [Online]. Available: https://doi.org/10.1145/2363.2366

[198] D. Giffhorn and G. Snelting, "A new algorithm for low-deterministic security," *International Journal of Information Security*, no. 14, pp. 263––287, 2015.

[199] F. Pottier, "A simple view of type-secure information flow in the spl pi-calculus," in *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, 2002, pp. 320–330.

[200] G. Barthe, J. M. Crespo, and C. Kunz, "Beyond 2-safety: Asymmetric product programs for relational program verification," in *Logical Foundations of Computer Science*, S. Artemov and A. Nerode, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–43.

[201] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Proceedings of the 12th International Conference on Static Analysis*, ser. SAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 352–367. [Online]. Available: https://doi.org/10.1007/11547662_24

[202] R. Joshi and K. R. M. Leino, "A semantic approach to secure information flow," *Sci. Comput. Program.*, vol. 37, no. 1–3, p. 113–138, May 2000. [Online]. Available:

https://doi.org/10.1016/S0167-6423(99)00024-6
[203] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings. 17th IEEE Computer Security Foundations Workshop*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2004, p. 100. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CSFW.2004.1310735
[204] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," *SIGPLAN Not.*, vol. 39, no. 1, p. 14–25, Jan. 2004. [Online]. Available: https://doi.org/10.1145/982962.964003
[205] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Proceedings of the 17th International Conference on Formal Methods*, ser. FM'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 200–214.
[206] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62.
[207] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin, "Dependent types and multi-monadic effects in F*," in *Principles of Programming Languages (POPL)*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 256–270.
[208] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 917–934.
[209] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370.
[210] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1807–1823. [Online]. Available: https://doi.org/10.1145/3133956.3134078
[211] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida
[212] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: A survey of fully abstract compilation and related work," *ACM Comput. Surv.*, vol. 51, no. 6, Feb. 2019. [Online]. Available: https://doi.org/10.1145/3280984
[213] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 256–271. [Online]. Available: https://doi.org/10.1109/CSF.2019.00025
[214] J. B. Jensen, N. Benton, and A. Kennedy, "High-level separation logic for low-level code," *SIGPLAN Not.*, vol. 48, no. 1, p. 301–314, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2480359.2429105
[215] J. C. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Pro-*

ceedings 17th Annual IEEE Symposium on Logic in Computer Science, 2002, pp. 55–74.

[216] A. Chlipala, "From network interface to multithreaded web applications: A case study in modular program verification," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 609–622. [Online]. Available: https://doi.org/10.1145/2676726.2677003

[217] P. Wang, S. Cuellar, and A. Chlipala, "Compiler verification meets cross-language linking via data abstraction," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds. ACM, 2014, pp. 675–690. [Online]. Available: https://doi.org/10.1145/2660193.2660201

[218] C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. Basin, "Igloo: Soundly linking compositional refinement and separation logic for distributed system verification," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428220

[219] S. Maus, M. Moskal, and W. Schulte, "Vx86: x86 assembler simulated in c powered by automated theorem proving," in *Algebraic Methodology and Software Technology*, J. Meseguer and G. Roşu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 284–298.

[220] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42.

[221] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 415–429.

[222] Y. Zhao, D. Sanán, F. Zhang, and Y. Liu, "High-assurance separation kernels: A survey on formal methods," *CoRR*, vol. abs/1701.01535, 2017. [Online]. Available: http://arxiv.org/abs/1701.01535

[223] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in *Operating Systems Design and Implementation (OSDI)*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 653–669.

[224] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying security invariants in ExpressOS," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 293–304, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451148

[225] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, "Jitk: A trustworthy in-kernel interpreter infrastructure," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 33–47.

[226] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, p. 107–115, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[227] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the FSCQ file system," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 18–37. [Online]. Available: https://doi.org/10.1145/2815400.2815402

[228] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser, "Cogent: Verifying

high-assurance file system implementations," *SIGPLAN Not.*, vol. 51, no. 4, p. 175–188, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2954679.2872404

[229] J. Yang and C. Hawblitzel, "Safe to the last instruction: Automated verification of a type-safe operating system," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 99–110. [Online]. Available: https://doi.org/10.1145/1806596.1806610

[230] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *Operating Systems Design and Implementation (OSDI)*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 165–181.

[231] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[232] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 351–366, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294295

[233] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. USA: IEEE Computer Society, 2014. [Online]. Available: https://doi.org/10.1109/SP.2014.26

[234] S. Fahl, *The Cyber Security Body of Knowledge*. University of Bristol, 2021, ch. Web & Mobile Security, version 1.0.1. [Online]. Available: https://www.cybok.org/

[235] M. Bugliesi, S. Calzavara, and R. Focardi, "Formal methods for web security," *Journal of Logical and Algebraic Methods in Programming*, vol. 87, pp. 110 – 126, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352220816301055

[236] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *ECOOP 2010 – Object-Oriented Programming*, T. D'Hondt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 126–150.

[237] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 87–100. [Online]. Available: https://doi.org/10.1145/2535838.2535876

[238] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to JavaScript," *SIGPLAN Not.*, vol. 48, no. 1, p. 371–384, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2480359.2429114

[239] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in javascript and its APIs," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1663–1671. [Online]. Available: https://doi.org/10.1145/2554850.2554909

[240] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[241] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-Wasm: Type-driven secure

cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290390

[242] A. Bohannon and B. C. Pierce, "Featherweight Firefox: Formalizing the core of a web browser," in *USENIX Conference on Web Application Development (WebApps 10)*. USENIX Association, Jun. 2010. [Online]. Available: https://www.usenix.org/conference/webapps-10/featherweight-firefox-formalizing-core-web-browser

[243] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and L. Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps," in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA, October 27, 2008*, V. Shmatikov, Ed. ACM, 2008, pp. 1–10. [Online]. Available: https://doi.org/10.1145/1456396.1456397

[244] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *2010 23rd IEEE Computer Security Foundations Symposium*, 2010, pp. 290–304.

[245] D. Fett, R. Küsters, and G. Schmitz, "An expressive model for the web infrastructure: Definition and application to the browser ID SSO system," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 673–688.

[246] B. C. Pierce, "The science of deep specification (keynote)," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, E. Visser, Ed. ACM, 2016, p. 1. [Online]. Available: https://doi.org/10.1145/2984043.2998388

[247] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1202–1219.

[248] W. Mansky, A. W. Appel, and A. Nogin, "A verified messaging system," *PACMPL*, vol. 1, no. OOPSLA, pp. 87:1–87:28, 2017. [Online]. Available: https://doi.org/10.1145/3133911

[249] N. Koh, Y. Li, Y. Li, L. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic, "From C to interaction trees: specifying, verifying, and testing a networked server," in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, A. Mahboubi and M. O. Myreen, Eds. ACM, 2019, pp. 234–248. [Online]. Available: https://doi.org/10.1145/3293880.3294106

[250] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "VST-Floyd: A separation logic tool to verify correctness of C programs," *J. Autom. Reasoning*, vol. 61, no. 1-4, pp. 367–422, 2018. [Online]. Available: https://doi.org/10.1007/s10817-018-9457-5

[251] C. Cotrini, T. Weghorn, D. Basin, and M. Clavel, "Analyzing first-order role based access control," in *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*. IEEE, 2015, pp. 3–17.

[252] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 196–205. [Online]. Available: https://doi.org/10.1145/1062455.1062502

[253] N. Zhang, M. Ryan, and D. P. Guelev, "Synthesising verified access control systems in XACML," in *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, ser. FMSE '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 56–65. [Online]. Available: https://doi.org/10.1145/1029133.1029141

[254] D. Gollmann, *The Cyber Security Body of Knowledge*. University of Bristol, 2021,

ch. Authentication, Authorisation & Accountability, version 1.0.2. [Online]. Available: https://www.cybok.org/

[255] N. Li and M. V. Tripunitara, "Security analysis in role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 4, p. 391–420, Nov. 2006. [Online]. Available: https://doi.org/10.1145/1187441.1187442

[256] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough, "Towards formal verification of role-based access control policies," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 242–255, 2008.

[257] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave tool for firewall analysis," in *Proceedings of the 24th International Conference on Large Installation System Administration*, ser. LISA'10. USA: USENIX Association, 2010, p. 1–8.

[258] C. Bodei, P. Degano, L. Galletta, R. Focardi, M. Tempesta, and L. Veronese, "Language-independent synthesis of firewall policies," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, 2018, pp. 92–106.

[259] C. Bertolissi, D. R. dos Santos, and S. Ranise, "Automated synthesis of run-time monitors to enforce authorization policies in business processes," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 297–308. [Online]. Available: https://doi.org/10.1145/2714576.2714633

[260] D. Basin, S. Burri, and G. Karjoth, "Dynamic enforcement of abstract separation of duty constraints," in *14th European Symposium on Research in Computer Security (ESORICS)*, ser. 5789, M. Backes and P. N. (Eds.), Eds., vol. 5789. Saint Malo, France: Springer-Verlag, 09 2009, pp. 250–267. [Online]. Available: http://www.springer.com/computer/security+and+cryptology/book/978-3-642-04443-4

ACRONYMS

ABAC Attribute-Based Access Control.

ACL2 A Computational Logic for Applicative Common Lisp.

AJAX Asynchronous JavaScript And XML.

API Application Programming Interface.

CCA Chosen Ciphertext Attack.

CCS Calculus of Communicating Systems.

CFI Control-Flow Integrity.

CSL Continuous Stochastic Logic.

CSP Communicating Sequential Processes.

CTL Computational Tree Logic.

CVC Cooperating Validity Checker.

DNS Domain Name System.

DOM Document Object Model.

EAL Evaluation Assurance Level.

FCF Foundational Cryptography Framework.

FDH Full Domain Hash.

FDR Failures-Divergences Refinement.

FORBAC First-Order Role-Based Access Control.

FPGA Field Programmable Gate Array.

GCC GNU Compiler Collection.

HDL Hardware Description Language.

HOL Higher Order Logic.

HRU Harrison, Ruzzo, and Ullman model.

HTTP Hyper Text Transport Protocol.

IEC International Electrotechnical Commission.

IFC Information Flow Control.

IND Indistinguishable.

ISO International Organization for Standardization.

JIT Just-In-Time.

JOANA Java Object-sensitive ANAlysis.

KA Knowledge Area.

KISS Knowledge In Security protocolS.

LTL Linear Temporal Logic.

LTS Labelled Transition System.

MAC Message Authentication Code.

NAT Network Address Translation.

OAEP Optimized Asymmetric Encryption Padding.

OS Operating System.

PCTL Probabilistic Computation Tree Logic.

PIN Personal Identification Number.

PKCS Public Key Cryptography Standards.

PSPACE Polynomial Space.

RBAC Role-Based Access Control.

RSA Rivest-Shamir-Adleman.

RTL Register Transfer Level.

SAML Security Assertion Markup Language.

SAT Satisfiability.

SET Secure Electronic Transactions.

SMT Satisfiability Modulo Theories.

SQL Structured Query Language.

SSO Single Sign-On.

SVA Secure Virtual Architecture.

TAL Typed Assembly Language.

TLA Temporal Logic of Actions.

TLS Transport Layer Security.

VCC Verifying C Compiler.

VST Verified Software Toolchain.

XACML Extensible Access Control Markup Language.

XML Extensible Markup Language.

GLOSSARY

CyBOK Refers to the Cyber Security Body of Knowledge.

INDEX