# SoK: Computer-Aided Cryptography

Manuel Barbosa*, Gilles Barthe†‡, Karthik Bhargavan§, Bruno Blanchet§, Cas Cremers¶, Kevin Liao†, Bryan Parno‖
*University of Porto (FCUP) and INESC TEC, †Max Planck Institute for Security & Privacy, ‡IMDEA Software Institute,
§INRIA Paris, ¶CISPA Helmholtz Center for Information Security, ‖Carnegie Mellon University

***Abstract*—Computer-aided cryptography is an active area of research that develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography. We present a cross-cutting systematization of the computer-aided cryptography literature, focusing on three main areas: (*i*) design-level security (both symbolic security and computational security), (*ii*) functional correctness and efficiency, and (*iii*) implementation-level security (with a focus on digital side-channel resistance). In each area, we first clarify the role of computer-aided cryptography—how it can help and what the caveats are—in addressing current challenges. We next present a taxonomy of state-of-the-art tools, comparing their accuracy and scope of analysis, trustworthiness, and usability. Then, we highlight their main achievements, trade-offs, and research challenges. After covering the three main areas, we present two case studies. First, we study efforts in combining tools focused on different areas to consolidate the guarantees they can provide. Second, we distill the lessons learned from the computer-aided cryptography community's involvement in the TLS 1.3 standardization effort. Finally, we conclude with recommendations to paper authors, tool developers, and standardization bodies moving forward.***

## I. Introduction

Designing, implementing, and deploying cryptographic mechanisms is notoriously hard to get right, with high-profile design flaws, devastating implementation bugs, and side-channel vulnerabilities being regularly found even in well-studied mechanisms. Each step is highly involved and fraught with pitfalls. At the design level, cryptographic mechanisms must achieve specific security goals against some well-defined class of attackers. Typically, this requires composing a series of sophisticated building blocks—abstract constructions for primitives, primitives for protocols, and protocols for systems. At the implementation level, high-level designs are then fleshed out with concrete functional details, such as data formats, session state, and programming interfaces. Moreover, implementations must be optimized for interoperability and performance. At the deployment level, implementations must also account for low-level threats that are absent at the design level, such as side-channel attacks.

Attackers are thus presented with a vast attack surface: They can break high-level designs, exploit implementation bugs, recover secret material via side-channels, or any combination of the above. Preventing such varied attacks on complex cryptographic mechanisms is a challenging task, and existing methods are hard-pressed to do so. Pen-and-paper security proofs often consider pared-down cryptographic cores of mechanisms to simplify analysis, yet remain highly complex and error-prone; demands for aggressively optimized implementations greatly increase the risks of introducing safety and correctness bugs, which are difficult to catch by code testing/auditing alone; ad-hoc constant-time coding recipes for mitigating side-channel attacks are tricky to implement, and yet may not cover whole gamut of leakage channels exposed in deployment. Unfortunately, the current modus operandi—relying on a select few cryptography experts armed with rudimentary tooling to vouch for security and correctness—simply cannot keep pace with the rate of innovation and development in the field.

*Computer-aided cryptography*, or CAC for short, is an active area of research that aims to address these challenges. Computer-aided cryptography encompasses formal, machine-checkable approaches to designing, analyzing, and implementing cryptography. The variety of tools available address different parts of the problem space. At the design level, tools can help manage the complexity of security proofs and even reveal subtle flaws or as-yet-unknown attacks in the process. At the implementation level, tools can guarantee that highly optimized implementations behave according to their functional specifications on all inputs. At the deployment level, tools can check that implementations are correctly protected against certain classes of side-channel attacks. Although individual tools may only address part of the problem, when combined, they can provide a high degree of assurance.

Computer-aided cryptography has already fulfilled some of these promises in focused but impactful settings. For instance, computer-aided security analyses had deep influence in the recent standardization of TLS 1.3 [1]–[4], and formally verified primitives are now being deployed at Internet-scale—HACL* [5] in Mozilla Firefox's NSS security engine and Fiat Cryptography [6] in Google's BoringSSL library. In light of these successes, there is growing enthusiasm for computer-aided cryptography. This is reflected in the rapid emergence of a dynamic community that brings together theoretical cryptographers, cryptography engineers, and formal method practitioners. Together, the community aims to achieve broader adoption of computer-aided cryptography, blending ideas from both fields, and more generally, to contribute to the future development of cryptography.

At the same time, computer-aided cryptography runs the risk of falling victim of its own success. Trust in the field can be undermined by the difficulty of understanding the guarantees of computer-aided cryptography artifacts and their fine-print caveats. For example, it has been asked whether the Selfie attack [7] contradicts prior claims of computer-aided cryptography proofs of TLS 1.3. The attack is an edge case of TLS 1.3's vast range of possible configurations, not covered by prior analysis, and therefore does not contradict, or diminish the value of, formal analyses that prove the absence of a large class of attacks. In addition, the field is increasingly broad, complex, and rapidly evolving, so no one has a complete understanding of every facet. This can make it difficult for the field to develop and address pressing challenges, such as the expected transition to post-quantum cryptography and scaling from primitives and protocols to cryptographic systems.

Given these concerns, the purpose of this SoK is three-fold:

1) We clarify the current capabilities and limitations of computer-aided cryptography.
2) We present a taxonomy of computer-aided cryptography tools, highlighting their main achievements and important trade-offs between them.
3) We outline promising new directions for computer-aided cryptography and related areas.

We hope this will help non-experts better understand the field, point experts to opportunities for improvement, and showcase to stakeholders (e.g., standardization bodies) the many benefits of computer-aided cryptography.

### A. Structure of the Paper

The subsequent three sections expand on the role of computer-aided cryptography in three main areas. Section II covers how to establish *design-level security* guarantees, using both symbolic and computational approaches. Section III covers how to develop *functionally correct and efficient* implementations. Section IV covers how to establish *implementation-level security* guarantees, with a particular focus on protecting against digital side-channel attacks.

We begin each of these sections with a critical review of the topic, explaining why the considered guarantees are important, how current tools and techniques outside CAC may fail to meet these guarantees, how CAC can help, the fine-print caveats of using CAC, and necessary technical background. We then taxonomize the state of the art tools for meeting these guarantees. To do this, we identify criteria along four main categories: accuracy (A) of modeling/analysis, scope (S) of modeling/analysis, trust (T), and usability (U). For each criteria, we label them with one or more categories, explain their importance, and inline some light discussion about the tools. The ensuing discussion highlights broader points, such as main achievements, important takeaways, and research challenges. Finally, we end each section with references for further reading. Given the amount of material we wish to cover, we are unable to be exhaustive in each area, but we would still like to point to other relevant lines of work.

Sections V and VI then describe important case studies. Having described how CAC tools can address the challenges of a particular problem area, in our first case study (Section V), we examine how to combine tools and consolidate the guarantees they can provide. In our second case study (Section VI), we distill the lessons learned from the computer-aided cryptography community's involvement in the TLS 1.3 standardization effort.

Finally, in Section VII, we close out with recommendations to paper authors, tool developers, and standardization bodies on how best to move the field of computer-aided cryptography forward.

## II. Design-Level Security

In this section, we focus on the role of computer-aided cryptography in establishing design-level security guarantees. Over the years, two flavors of design-level security have been developed in two largely separate communities—symbolic security (in the formal methods community) and computational security (in the cryptography community). This has led to two complementary strands of work, so we aim to cover them both.

### A. Critical Review

***Why is design-level security important?*** Validating cryptographic designs through mathematical arguments is perhaps the only way to convincingly demonstrate that they are secure against entire classes of attacks. One class of attacks naturally captured in the symbolic model are attacks that exploit flaws in a protocol's logic. These range from basic man-in-the-middle or reflection attacks to complex attacks involving 18+ messages to drive the protocol into an insecure state [2], [8]. The computational model goes beyond the symbolic model, at the expense of more intricate proofs and less automation, by reasoning about the probability that an (often computationally bounded) attacker breaks a design. This applies to primitives as well as protocols and systems. For the latter, the computational model considers both attacks in the underlying cryptographic primitives and flaws that arise from their composition.

***How can design-level security fail?*** The current modus operandi of validating the security cryptographic designs using pen-and-paper arguments is alarmingly fragile. This is for two main reasons:

- *Erroneous arguments.* Writing security arguments is tedious and error-prone, even for experts. Because they are primarily done on pen-and-paper, errors are difficult to catch and can go unnoticed for years.
- *Inappropriate modeling.* Even when security arguments are correct, attacks can lie outside the model in which they are established. This is a known and common pitfall: To make (pen-and-paper) security analysis tractable, models are often heavily simplified into a cryptographic core that elides many details about cryptographic designs and attacker capabilities. Unfortunately, unaccounted attacks are often found outside of this core.

***How are these failures being addressed outside CAC?*** To minimize erroneous arguments, the game-based code-playing methodology [9] advocates decomposing security arguments into more elementary steps that are easier to understand and get right. However, pen-and-paper proofs based on this methodology remain error-prone, which has led to suggestions of using computer-aided tools [10].

To reduce the risks of inappropriate modeling, real-world provable security [11]–[13] advocates making security arguments in more accurate models of cryptographic designs and adversarial capabilities. Unfortunately, the added realism comes with greater complexity, which in turn complicates security analysis.

***How can computer-aided cryptography help?*** Computer-aided cryptography tools are effective for detecting flaws in cryptographic designs and for managing the complexity of security proofs. They crystalize the benefits of code-based game playing and of real-world provable security, and deliver trustworthy analyses for complex designs that are beyond reach of pen-and-paper analysis.

***What are the fine-print caveats?*** Computer-aided cryptography artifacts are only as good as their top-level statements. However, understanding these statements can be challenging, as most security proofs rely on implicit assumptions, e.g., adequacy of the model for the symbolic model, or intractability of computational problems and availability of a perfect source of randomness for the computational model. Without proper guidance, reconstructing top-level statements can be challenging, even for experts. (As an analogy, it is hard even for a talented mathematician to track all dependencies in a textbook.) Finally, as any software, tools may have bugs.

***What background do I need to know?*** The symbolic model has mostly been applied to cryptographic protocols, rather than non-interactive low-level primitives. This is because the goal of the symbolic model is to reduce the complexity of analyzing protocols by assuming the lower-level components are ideal (e.g., an adversary can only decrypt ciphertexts if it has knowledge of the *entire* secret key). This idealization ensures that security protocol can be modeled and verified using symbolic logic, which lends to automatically searching for and unveiling logical flaws in complex cryptographic protocols and systems.

The computational model has been applied to a range of cryptographic schemes, spanning primitives, protocols, and systems. Overwhelmingly, cryptographic designs are probabilistic, and security notions are modeled by probabilistic experiments, traditionally called games. A design is secure as long as security breaks happen with negligible probability (e.g., a signature scheme is unforgeable if an adversary has a negligible probability to forge a valid signature). Most proofs in the computational model are reductionist, and show that successful attacks can be turned into algorithms for solving computationally intractable problems. The quality of reductionist arguments depends on the choice of the target problems and the computational complexity (e.g., linear or polynomial) of the reduction. Computational proofs often decompose reasoning into elementary steps, and interleave steps about "hops" between probabilistic experiments (e.g., proving that an event is equiprobable in the two experiments), and steps about single probabilistic experiments (e.g., proving that an event has bounded probability).

Game-based proofs in the computational model do not scale well to complex cryptographic systems such as secure messaging or cloud computing. Moreover, a general problem with secure-design arguments is whether these hold in arbitrary contexts. To deal with these problems, compositional approaches to computational security proofs have also been proposed that permit using a divide-and-conquer approach. One such approach, that is widely used to reason about cryptographic protocols of varying complexity is based on the simulation paradigm [14].

| Tool | | Unbound | Eq-thy | State | Trace | Equiv | Link |
|---|---|---|---|---|---|---|---|
| CPSA$^{\triangleright}$ | [15] | ● | ○ | ● | ● | ○ | ○ |
| F7$^{\diamond}$ | [16] | ● | ◐ | ● | ● | ○ | ● |
| ↳F5$^{\diamond}$ | [17] | ● | ◐ | ● | ● | ○ | ● |
| Maude-NPA$^{\triangleright}$ | [18] | ● | ● | ○ | ● | ●$^{d}$ | ○ |
| ProVerif$^{\star\dagger}$ | [19] | ● | ◐ | ○ | ● | ●$^{d}$ | ○ |
| ↳fs2pv$^{\diamond}$ | [20] | ● | ◐ | ○ | ● | ○ | ● |
| ↳StatVerif$^{\star\dagger}$ | [21] | ● | ◐ | ● | ● | ●$^{d}$ | ○ |
| Scyther$^{\triangleright}$ | [22] | ● | ○ | ○ | ● | ○ | ○ |
| scyther-proof$^{\triangleright\ddagger\S}$ | [23] | ● | ○ | ○ | ● | ○ | ○ |
| Tamarin$^{*\ddagger}$ | [24] | ● | ● | ● | ● | ●$^{d}$ | ○ |
| ↳SAPIC$^{\star}$ | [25] | ● | ● | ● | ● | ●$^{d}$ | ○ |
| CI-AtSe$^{\triangleright}$ | [26] | ○ | ● | ● | ● | ○ | ○ |
| OFMC$^{\triangleright\dagger}$ | [27] | ○ | ◐ | ● | ● | ○ | ○ |
| SATMC$^{\triangleright}$ | [28] | ○ | ○ | ● | ● | ○ | ○ |
| AKISS$^{\star}$ | [29] | ○ | ● | ● | ○ | ●$^{t}$ | ○ |
| APTE$^{\star}$ | [30] | ○ | ○ | ● | ○ | ●$^{t}$ | ○ |
| DEEPSEC$^{\star}$ | [31] | ○ | ◐ | ● | ○ | ●$^{t,l}$ | ○ |
| SAT-Equiv$^{\star}$ | [32] | ○ | ○ | ○ | ○ | ●$^{t}$ | ○ |
| SPEC$^{\star}$ | [33] | ○ | ○ | ○ | ○ | ●$^{o}$ | ○ |

| Specification language | Miscellaneous symbols |
|---|---|
| $\triangleright$ – security protocol notation | ↳ – previous tool extension |
| $\star$ – $\pi$-calculus based | † – abstractions |
| $*$ – multiset rewriting | ‡ – interactive mode |
| $\diamond$ – general programming language | § – independent verifiability |

| Equational theories (Eq-thy) | Equivalence properties (Equiv) |
|---|---|
| ● – with AC axioms | $t$ – trace equivalence |
| ◐ – without AC axioms | $l$ – labeled bisimilarity |
| ○ – fixed | $o$ – open bisimilarity |
| | $d$ – diff equivalence |

TABLE I
OVERVIEW OF TOOLS FOR SYMBOLIC SECURITY ANALYSIS. SEE SECTION II-B FOR MORE DETAILS ON COMPARISON CRITERIA.

### B. *Symbolic Tools: State of the Art*

Table I presents a taxonomy of modern, general-purpose symbolic tools. Tools are listed in three groups (demarcated by dashed lines): unbounded trace-based tools, bounded trace-based tools, and equivalence-based tools; within each group, top-level tools are listed alphabetically. Tools are categorized as follows.

***Unbounded number of sessions (A).*** Can the tool analyze an unbounded number of protocol sessions? There exist protocols that are secure when at most $N$ sessions are considered, but become insecure with more than $N$ sessions [34]. Tools that explicitly limit the analyzed number of sessions perform bounded analysis (○): They do not consider attacks beyond the cut-off. Tools that offer unbounded analysis (●) can prove the absence of attacks within the model, but at the cost of undecidability [35]. However, in practice, modern unbounded tools typically substantially outperform bounded tools even for a small number of sessions, and therefore enable the analysis of more complex models.

***Equational theories (S).*** Equational theories capture mathematical identities that hold in the underlying model. These are important for broadening analysis—ignoring valid equations implicitly weakens the class of adversaries considered. All tools support equational theories, but the range of equational theories supported varies. The finer details often make them incomparable, and even where they overlap, they are not all equally effective for analyzing concrete protocols. We provide a coarse classification: Fixed (○), without associative-commutative (AC) axioms (◐), and with AC axioms (●). At a high-level, extra support for axioms enables detecting a larger class of attacks, see, e.g., [36], [37].

***Global mutable state (S).*** Does the tool support verification of protocols with global mutable state? Many real-world protocols involve databases, registers, key servers, making support for global mutable state crucial for considering complex attack scenarios [25].

***Trace properties (S).*** Does the tool support verification of trace properties? Trace properties state that a bad event never occurs on any execution trace. Secrecy and authentication are prime examples of trace properties. For example, a protocol preserves secrecy if, for any execution trace, secret data is absent from adversarial knowledge.

***Equivalence properties (S).*** Does the tool support verification of equivalence properties? Equivalence properties capture that an adversary is unable to distinguish between two executions. Indistinguishability-based secrecy, unlinkability, anonymity are prime examples. Equivalence properties capture security notions that cannot (naturally or precisely) be expressed as trace properties. They are inherently more difficult to verify than trace properties, because they involve relations between traces instead of single traces, and tool support for such properties is substantially less mature than for trace properties. There are several notions of equivalence: trace ($t$), labeled bisimilarity ($l$), open bisimilarity ($o$), and diff equivalence ($d$). For lack of space, we only discuss the two most used notions (see Section II-C). For a more formal treatment of all these notions, see the survey by Delaune and Hirschi [38].

***Link to implementation (T).*** Can the tool extract/generate executable code from specifications in order to link symbolic security guarantees to implementations?

† ***Abstractions (U).*** Does the tool use abstraction? Algorithms may use abstraction to overestimate attack possibilities, e.g., by computing a superset of the adversary's knowledge. This can yield more efficient and fully automatic analysis systems and can be a workaround to undecidability, but comes at the cost of incompleteness, i.e., false attacks may be found or the tool may terminate with an indefinite answer.

‡ ***Interactive mode (U).*** Does the tool support an interactive analysis mode? Interactive modes generally trade off automation for control. While push-button tools are certainly desirable, they may fail opaquely (perhaps due to undecidability barriers), leaving it unclear or impossible to proceed. Interactive modes can allow users to analyse failed automated analysis attempts, inspect partial proofs, and to provide hints and guide analyses to overcome any barriers.

§ ***Independent verifiability (T).*** Are the analysis results independently machine-checkable? Symbolic tools implement complex verification algorithms and decision procedures, which may be buggy and return incorrect results. This places them in the trusted computing base. The one exception is scyther-proof [23], which generates proof scripts that can be machine-checked in the Isabelle theorem prover [39].

***Specification language (U).*** How are protocols specified? The categorizations are domain-specific security protocol languages (▷), $\pi$-calculus based ($\star$), multiset rewriting ($*$), and general programming language ($\diamond$). Their differences are too nuanced to describe here, but interested readers should refer to the cited tool papers for more information.

### *C. Symbolic Security: Discussion*

***Achievements: Symbolic proofs for real-world case studies.*** The applications of symbolic tools are too vast to survey here, but ProVerif and Tamarin stand out as having been used to analyze large, real-world protocols. ProVerif has been used to analyze TLS 1.0 [40], TLS 1.3 [3], Signal [41], and Noise protocols [42]. Tamarin has been used to analyze the 5G authentication key exchange protocol [43], TLS 1.3 [2], [4], and the DNP3 SAv5 power grid protocol [44]. ProVerif and Tamarin offer unprecedented combinations of scalability and expressivity, which enables them to deal with complex systems and properties.

***Challenge: Verifying equivalence properties.*** Many security properties can be modeled accurately by equivalence properties, but their verification is limited: Either one bounds the number of sessions or one has to use the very strong notion of diff-equivalence, which cannot handle many desired properties, e.g., vote privacy in e-voting and unlinkability. Diff-equivalence, first introduced in ProVerif [45] and later adopted by Maude-NPA [46] and Tamarin [47], remains the only fully automated approach for proving equivalences for an unbounded number of sessions. However, trace equivalence is arguably the most adequate for formalizing privacy properties. For the bounded setting, recent developments include more support for more equational theories (AKISS, DEEPSEC), for protocols with else branches (APTE, AKISS, DEEPSEC) and for protocols whose inputs are not entirely determined by their inputs (APTE, DEEPSEC). There have also been performance improvements based on partial order reduction (APTE, DEEPSEC) or graph planning (SAT-Equiv). Still, verifying general equivalence properties for an unbounded number of sessions remains a challenge.

### *D. Computational Tools: State of the Art*

Table II presents a taxonomy of general-purpose computational tools. Tools are listed alphabetically and are categorized as follows.

***Automation (U).*** All tools provide some sort of automation, so here we single out tools that can automatically find security proofs.

***Composition (U).*** Does the tool support decomposing security arguments for cryptographic systems into security arguments for their core components? Compositional reasoning is essential to guarantee scalable analysis.

| Tool | | Auto | Comp | CS | GH | Unary | Link | TCB |
|---|---|---|---|---|---|---|---|---|
| AutoG&P$^{\diamond}$ | [48] | ● | ○ | ◐ | ◓ | ◓ | ○ | self, SMT |
| CertiCrypt$^{\triangleright\diamond}$ | [49] | ○ | ○ | ● | ◒ | ◒ | ● | Coq |
| CryptHOL$^{\diamond}$ | [50] | ○ | ● | ◐ | ◒ | ◒ | ○ | Isabelle |
| CryptoVerif$^{\star\diamond}$ | [51] | ● | ○ | ● | ◓ | ◓ | ● | self |
| EasyCrypt$^{\triangleright\diamond}$ | [52] | ○ | ● | ◐ | ◒ | ◒ | ● | self, SMT |
| F7$^{\diamond}$ | [16] | ○ | ● | ○ | ◓ | ◓ | ● | self, SMT |
| F$^{*\diamond}$ | [53] | ○ | ● | ○ | ◓ | ◓ | ● | self, SMT |
| FCF$^{\diamond}$ | [54] | ○ | ● | ◐ | ◒ | ◒ | ● | Coq |
| ZooCrypt$^{\diamond}$ | [55] | ● | ○ | ● | ◓ | ◓ | ○ | self, SMT |

| Concrete security (CS) | Game hopping (GH) |
|---|---|
| ● – security + efficiency | ◓ – automation focus |
| ◐ – security only | ◒ – expressiveness focus |
| ○ – no support | ○ – no support |

| Unary | Specification language |
|---|---|
| ◓ – automation focus | $\star$ – process calculus |
| ◒ – expressiveness focus | $\triangleright$ – imperative |
| ○ – no support | $\diamond$ – functional |

TABLE II
OVERVIEW OF TOOLS FOR COMPUTATIONAL SECURITY ANALYSIS. SEE SECTION II-D FOR MORE DETAILS ON COMPARISON CRITERIA.

***Concrete security (A).*** Can the tool be used to prove concrete bounds on success probability and attacker execution time? We consider tools with no such support (○), with support for concrete success probabilities only (◐) and for both (●).

***Game hopping (U).*** Is there support for game hopping, i.e., does the tool support common principles of game-based proofs (bridging steps, failure event steps, hybrid arguments) or not (○)? If so, is the emphasis put on automation (◓) or on being able to express arbitrary arguments (◒), i.e., on expressivity. We note that in F7 and F$^*$, game hopping is based on ideal functionalities and justified externally (see [56] for more information).

***Unary reasoning (U).*** Is there support for reasoning about strong invariants or probabilities of events over a single program execution or not (○)? If so, is the emphasis put on automation (◓), in which case the reasoning is usually only about deterministic yes/no properties, or on being able to express arbitrary probabilistic properties (◒), i.e., on expressivity.

***Link to implementation (T).*** Can the tool extract/generate executable code from specifications in order to link computational security guarantees to implementations?

***Trusted computing base (T).*** What lies in the trusted computing base (TCB)? A well established general-purpose theorem prover such as Coq or Isabelle is usually held as the minimum TCB for proof checking. Most tools, however, rely on an implementation of the tool logics in a general purpose language that must be trusted (self). Automation often relies on general-purpose SMT solvers.

***Specification language (U).*** What kind of specification language is used? All tools support some functional language core for expressing the semantics of operations ($\diamond$). Some tools support an imperative language ($\triangleright$) in which to write security games, while others rely on a process calculus ($\star$).

### *E. Computational Security: Discussion*

***Achievements: Machine-checked security for real-world cryptographic designs.*** Computational tools have progressed considerably over the years, to the point that they are now being used to verify the security of real-world cryptographic primitives, protocols, and systems. CryptoVerif has been used for a number of protocols, including TLS 1.3 [3], Signal [41], and WireGuard [57]. EasyCypt has been used for the Amazon Web Service (AWS) key management system [58] and the SHA-3 standard [59]. F7 was used to build miTLS, a reference implementation of TLS 1.2 with verified computational security at the code-level [60], [61]. F$^*$ was used to implement and verify the security of the TLS 1.3 record layer [1].

***Takeaway: CryptoVerif is good for highly automated computational analysis of protocols and systems.*** CryptoVerif is both a proof-finding and proof-checking tool. It works particularly well for protocols (e.g., key exchange), as it can produce automatically or with a light guidance a sequence of proof steps that establish security. One distinctive strength of CryptoVerif is its input language based on the applied $\pi$-calculus [62], which is well-suited to describing protocols that exchange messages in sequence. Another strength of CryptoVerif is a carefully crafted modeling of security assumptions that help the automated discovery of proof steps. In turn, automation is instrumental to deal with large cryptographic games and games that contain many different cases, as is often the case in proofs of protocols.

***Takeaway: F$^*$ is good for analysis of full protocols and systems.*** F$^*$ is a general-purpose verification-oriented programming language. It works particularly well for analyzing cryptographic protocols and systems beyond their crypographic core. Computational proofs in F$^*$ rely on transforming a detailed protocol description through a series of game transformations into a final (ideal) program by relying on ideal functionalities for cryptographic primitives. Formal validation of the intermediate transformations is carried out manually, with some help from the F$^*$ verification infrastructure. Formal verification of the final program is done fully within F$^*$. This approach is driven by the insight that critical security issues, and therefore also potential attacks, often arise only in detailed descriptions of full protocols and systems (compared to when reasoning about cryptographic cores). The depth of this insight is reflected by the success of F$^*$-based verification both in helping discovering new attacks on real-world protocols like TLS [8], [63] as well as in verifying their concrete design and implementation [1], [60].

***Takeaway: EasyCrypt is the closest to pen-and-paper cryptographic proofs.*** EasyCrypt supports game-hopping through a general-purpose relational program logic that captures many of the common game-hopping techniques, e.g., bridging, failure event, and reduction steps. This is complemented by libraries that support other common techniques, e.g., the PRF/PRP switching lemma, hybrid arguments, and lazy sampling. Overall, the game sequences in EasyCrypt proofs closely matches pen-and-paper arguments—when the latter are correct. A consequence is that EasyCrypt is amenable to proving security of primitives, as well as protocols and systems.

***Challenge: Scaling security proofs for cryptographic systems.*** Analyzing large cryptographic systems is best done in a modular way by composing simpler building blocks.

However, cryptographers have long recognized the difficulties of preserving security under composition [64]. Most game-based security definitions do not to provide out-of-the-box composition guarantees, so simulation-based definitions are the preferred choice for analyzing large cryptographic systems, with UC being the gold-standard—universally composable (UC) definitions guarantee secure composition in arbitrary contexts [65]. Work on developing machine-checked UC proofs is relatively nascent [66]–[68], but is an important and natural next step for computational tools.

### *F. Further Reading*

We provide pointers to relevant developments not covered in this section. Several tools leverage the benefits of automated verification to support automated synthesis of secure cryptographic designs, mainly in the computational world [55], [69]–[72]. Cryptographic compilers have been proposed for verifiable computation [73]–[76], zero-knowledge [77]–[80], and secure multiparty computation [81] protocols, which are parametrized by a proof-goal or a functionality to compute. Some of these compilers are supported by proofs that guarantee that the output protocols are correct and/or secure for every input specification [82]–[85]. We recommend readers to also consult other recent surveys in the field. Blanchet [86] surveys design-level security until 2012 (with a focus on ProVerif). Cortier et al. [87] survey computational soundness results, which transfer security properties from the symbolic world to the computational world.

## III. Functional Correctness and Efficiency

In this section, we focus on the role of computer-aided cryptography in developing functionally correct and efficient implementations.

### *A. Critical Review*

***Why are functional correctness and efficiency important?*** To reap the benefits of design-level security guarantees, concrete implementations must be an accurate translation of the design proven secure. That is, they must be functionally correct (i.e., have equivalent input/output behavior) with respect to the design specification. Moreover, to meet practical deployment requirements, implementations must be efficient. Cryptographic routines are often on the critical path for security applications (e.g., for reading and writing TLS packets or files in an encrypted file system), and so even a few additional clock-cycles can have a detrimental impact on system performance.

***How can functional correctness and efficiency fail?*** If performance is not an important goal, then achieving functional correctness is relatively easy—just use a reference implementation that does not deviate too far from the specification, so that correctness is straightforward to argue. However, performance demands drive cryptographic code into extreme contortions that make functional correctness difficult to achieve, let alone prove. For example, OpenSSL is one of the fastest open source cryptographic libraries; they achieve this speed in part through the use of Perl code to generate strings of text that additional Perl scripts interpret to produce input to the C preprocessor, which ulimately produces highly tuned, platform-specific assembly code [94]. Many more examples of high-speed crypto code written at assembly and pre-assembly levels can be found in SUPERCOP [98], a benchmarking framework for cryptography implementations.

More broadly, efficiency considerations typically rule out using high-level languages. Instead, C and assembly are the de facto tools of the trade, adding memory safety to the list of important requirements. Indeed, memory errors can compromise secrets held in memory, e.g., in the Heartbleed attack [99]. Fortunately, as we discuss below, proving memory safety is table stakes for most of the tools we discuss. Additionally, achieving best-in-class performance demands aggressive, platform-specific optimizations, far beyond what is achievable by modern optimizing compilers (which are problematic in their own ways, as we will see in Section IV). Currently, these painstaking efforts are manually repeated for each target architecture.

***How are these failures being addressed outside CAC?*** Given its difficulty, the task of developing high-speed cryptography is currently entrusted to a handful of experts. Even so, experts make mistakes (e.g., a performance optimization to OpenSSL's AES-GCM implementation nearly reached deployment even though it enabled arbitrary message forgeries [100]; an arithmetic bug in OpenSSL led to a full key recovery attack [101]), and the current solutions for preventing more of them are (1) auditing, which is costly in both time and expertise, and (2) testing, which cannot be complete for the size of inputs used in cryptographic algorithms. These solutions are also clearly inadequate: Despite widespread usage and scrutiny, OpenSSL's cryptographic library `libcrypto` reported 24 vulnerabilities between January 1, 2016 and May 1, 2019 [102].

***How can computer-aided cryptography help?*** Cryptographic code is an ideal target for program verification. Such code is both critically important and difficult to get right. The use of heavyweight formal methods is perhaps the only way to attain the high-assurance guarantees expected of them. At the same time, because the volume of code in cryptographic libraries is relatively small (compared to, say, an operating system), verifying complex, optimized code is well within reach of existing tools and reasonable human effort. And indeed, verified cryptographic primitives are already outperforming their unverified counterparts, as we will see shortly.

***What are the fine-print caveats?*** Functional correctness makes implicit assumptions, e.g., correct modeling of hardware functional behavior. Another source of implicit assumptions is the gap between code and verified artifacts, e.g., verification is carried on source code, or on a verification-friendly representation. Moreover, proofs may presuppose correctness of libraries, e.g., for efficient arithmetic. Finally, as with any software, verification tools may have bugs.

***What background do I need to know?*** Functional correctness is the central focus of program verification. An

| Tool | | Memory safety | Automation | Parameter-ized | Input language | Target(s) | TCB |
|---|---|---|---|---|---|---|---|
| Cryptol + SAW | [88] | ● | ◐ | ○ | C, Java | C, Java | SAT, SMT |
| CryptoLine | [89] | ○ | ● | ○ | CryptoLine | C | Boolector, MathSAT, Singular |
| Dafny | [90] | ● | ◐ | ○ | Dafny | C#, Java, JavaScript, Go | Boogie, Z3 |
| F* | [53] | ● | ◐ | ○ | F* | OCaml, F#, C, Asm, Wasm | Z3, typechecker |
| Fiat Crypto | [6] | ● | ○ | ● | Gallina | C | Coq, C compiler |
| Frama-C | [91] | ● | ◐ | ○ | C | C | Coq, Alt-Ergo, Why3 |
| gfverif | [92] | ○ | ● | ○ | C | C | g++, Sage |
| Jasmin | [93] | ● | ◐ | ○ | Jasmin | Asm | Coq, Dafny, Z3 |
| Vale | [94], [95] | ● | ◐ | ● | Vale | Asm | Dafny or F*, Z3 |
| VST | [96] | ● | ○ | ○ | Gallina | C | Coq |
| Why3 | [97] | ○ | ◐ | ○ | WhyML | OCaml | SMT, Coq |

Automation
● – automated ◐ – automated + interactive ○ – interactive

TABLE III
OVERVIEW OF TOOLS FOR FUNCTIONAL CORRECTNESS. SEE SECTION III-B FOR MORE DETAILS ON COMPARISON CRITERIA.

implementation can be proved functionally correct in two different ways: equivalence to a reference implementation, or satisfying a functional specification, typically expressed as pre-conditions (what the program requires on inputs) and post-conditions (what the program guarantees on outputs). Both forms of verification are supported by a broad range of tools. A unique aspect of cryptographic implementations is that corrrectness proofs often rest on non-trivial mathematics, and therefore require striking a good balance between automation and user control. Nevertheless, SMT-based automation remains instrumental for minimizing verification effort, and almost all tools offer an SMT-based backend.

Functional correctness is overwhelmingly carried out at source level. A long-standing challenge, then, is how to carry guarantees to machine code. This can be addressed using verified compilers, which are supported by a formal correctness proof. CompCert [103] is a prime example of moderately optimizing verified compiler for a large fragment of C. However, the trade-off is that verified compilers typically come with fewer optimizations than mainstream compilers.

### B. Program Verification Tools: State of the Art

Table III presents a taxonomy of program verification tools that have been used for cryptographic implementation. Tools are listed alphabetically and are categorized as follows.

***Memory-safety (S).*** Can the tool verify that programs are memory safe? Memory safety ensures that all runs of a program are free from memory errors (e.g., buffer overflow, null pointer dereferences, use after free).

***Automation (U).*** Tools provide varying levels of automation. We give a coarse classification: automatic tools (●), tools that combine automated and interactive theorem proving (◐), and tools that allow only interactive theorem proving (○).

***Parameterized (U).*** Can the tool implement and verify parameterized code? This enables writing and verifying generic code that can be used to produce different implementations depending on the supplied parameters. For example, Fiat Crypto can generate elliptic curves implementations parameterized by a prime modulus; Vale implementations are parameterized by OS, assembler, and hardware platform.

***Input language (U).*** What is the input language? Many toolchains use custom verification-oriented languages, such as Dafny, F*, Gallina, Jasmin, CryptoLine, and WhyML. Others take code written in existing languages (e.g., C, Java) as input.

***Target(s) (A,S).*** At what level is the analysis carried out (e.g., source-level or assembly-level)? Note that tools targeting source-level analysis must use verified compilers (e.g., CompCert [103]) to carry guarantees to machine-level, which comes with a performance penalty. Tools targeting assembly-level analysis sidestep this dilemma, but generally verification becomes more difficult.

***Trusted computing base (T).*** What lies in the trusted computing base? Many verification frameworks rely on untrusted, building-block verification tools, such as SMT solvers (e.g., Z3) and interactive theorem provers (e.g., Coq). While these are acknowledged to be important trust assumptions of verification tools, verified artifacts tend to rely on additional trust assumptions, e.g., unverified interoperability between tools or only verifying small routines in a larger primitive.

### C. Discussion

***Achievements: Verified primitives are being deployed at Internet-scale.*** A recent milestone achievement of computer-aided cryptography is that verified primitives are finally being deployed at scale. Verified primitives in the HACL* [5] library have made their way into Mozilla Firefox's NSS security engine, and verified elliptic curve implementations in the Fiat Cryptography library [6] have made their way into Google's BoringSSL library.

There are several common insights to their success. First, verified code needs to be as fast or faster than the code being replaced. Second, verified code needs to fit the APIs that are actually in use. Third, it helps if team members work with or take internships with the companies that take the code. In the case of HACL*, it additionally helped that they replaced an entire ciphersuite, and that they were willing to undertake a significant amount of non-research work, such as packaging and testing, that many academic projects stop short of.

***Takeaway: Verified implementations are now as fast or faster than their unverified counterparts.*** Through decades of research in formal verification, it was commonly accepted that the proof burden in verifying complex, optimized code was exorbitant; verified code would be hard-pressed to compete with unverified code in terms of performance. Recent advances

| Implementation | | FC | CC | Tool(s) | Target | % Faster |
|---|---|---|---|---|---|---|
| evercrypt | [102] | ● | ● | F*, Vale | 64-bit C, Intel ADX asm | 25.92 |
| precomp | [104] | ○ | ○ | – | Intel ADX asm | 25.77 |
| sandy2x | [105] | ○ | ○ | – | Intel AVX asm | 11.15 |
| hacl | [102] | ● | ● | F* | 64-bit C | 8.69 |
| jasmin | [93] | ○ | ● | Jasmin | Intel x86_64 asm | 7.88 |
| amd64 | [106] | ◐ | ○ | Coq, SMT | Intel x86_64 asm | 6.11 |
| fiat | [6] | ● | ○ | Fiat Crypto | 64-bit C | 5.39 |
| donna64 | [107] | ○ | ○ | – | 64-bit C | 0.00 |

Functional correctness (FC), Constant-time (CC)

● – verified ◐ – partially verified ○ – not verified

TABLE IV

COMPARISON OF CURVE25519 IMPLEMENTATIONS. % FASTER CALCULATED USING DONNA64 AS THE BASELINE.

in various other domains have challenged this position, producing verified implementations with competitive performance to unverified implementations (e.g., IronFleet [108]). Now, in the cryptography domain, we are seeing verified implementations that meet and exceed the performance of some of the fastest unverified implementations.

As a small case study, we look at Curve25519 [109], a widely used elliptic curve that has received considerable interest from the applied cryptography community (in setting new speed records) and the formal methods community (in verifying that high-speed implementations are correct and secure). We compare a number of Curve25519 implementations in Table IV. These comprise some of the fastest available verified and unverified implementations; they are written in C, assembly, or a combination of both.

To see how they stack up in terms of efficiency, we measure the number of CPU cycles (median over 5K executions) it takes to perform scalar multiplication. We report the performance increase (% Faster) over donna64 [107], one of the fastest known (unverified) C implementations. All measurements are collected on a 1.8 GHz Intel i7-8565U with 16 GB of RAM; hyperthreading and dynamic-processor scaling (e.g., Turbo Boost) are disabled. Implementations written in C are compiled using GCC 9.2 with flag -O3. To summarize, verified implementations beat unverified implementations in both portable C implementations and assembly implementations.

***Takeaway: Higher performance entails larger verification effort.*** Verifying generic, high-level code is typically easier, but comes with a performance cost. Hand-written assembly can achieve best in class performance by taking advantage of hardware-specific optimizations, but verifying such implementations is quite difficult due to complex side-effects, unstructured control-flow, and flat structure. Moreover, this effort must be repeated for each platform. C code is less efficient, as hardware-specific features are not a part of standard portable C, but implementations need only be verified once and can then be run on any platform. Code written in higher-level languages is even less efficent, but verification becomes much easier (e.g., memory safety can be obtained for free). These aspects are discussed in more detail in the Vale and Jasmin papers [93], [94], [110].

***Challenge: Automated "game-hopping" equivalence proofs.*** Significant progress could be made if functional correctness proofs could be solved by providing a sequence of simple transformations that connect spec to target and relying on an automatic tool to check these simple transformations. Promising recent work in this direction [111] demonstrates the feasibility of the approach. However, the current approaches are not automatic: neither in finding the "hops" nor in proving the hops. The latter seems achievable for many useful control-flow-preserving transformations, whereas the former could be feasible at least for common control-flow transformations.

***Challenge: Functional correctness of common arithmetic routines.*** Verifying cryptographic code inevitably involves a variety of tricky mathematical reasoning that even SMT-based tools can struggle with. Examples range from proving the correctness of the Montgomery representations used to accelerate big-integer computations, to the nuts-and-bolts of converting between, say, 64-bit words and the underlying bytes. At present, most verification efforts build this infrastructure from scratch and customize it for their own particular needs, which leads to significant duplication of effort across projects. Hence, an open challenge is to devise a common core of such routines (e.g., a verified version of the GMP library [112]) that can be shared across all (or most) verification projects, despite their reliance on different tools and methodologies.

### *D. Further Reading*

While our principal focus is on cryptographic code, verifying systems code is an important and active area of research. For example, there has been significant work in verifying operating systems code [113]–[119], distributed systems [108], [120], [121], and even entire software stacks [122]. We expect that these two strands of work will cross paths in the future.

## IV. IMPLEMENTATION-LEVEL SECURITY

In this section, we focus on the role of computer-aided cryptography in establishing implementation-level security guarantees, with a particular focus on protecting against digital side-channel attacks. By digital side-channel attacks, we mean those that can be lauched by observing intentionally exposed interfaces by the computing platform, including all execution time variations and observable side-effects in shared resources such as the cache. This excludes physical side channels such as power consumption, electromagnetic radiation, etc.

### *A. Critical Review*

***Why is implementation-level security important?*** Although design-level security can rule out large classes of attacks, guarantees are proven in a model that idealizes an attacker's interface with the underlying algorithms: They can choose inputs and observe outputs. However, in practice, attackers can observe much more than just the functional behavior of cryptographic algorithms. For example, side-channels are interfaces available at the implementation-level (but unaccounted for at the design-level) from which information can leak as side-effects of the computation process (e.g., timing behavior, memory access patterns). And indeed, these sources of leakage

| Tool | | Target | Method | Synthesis | Sound | Complete | Public inputs | Public outputs | Control flow | Memory access | Variable-time op. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ABPV13 | [123] | C | DV | ○ | ● | ● | ● | ○ | ● | ● | ○ |
| CacheAudit | [124] | Binary | Q | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ |
| ct-verif | [125] | LLVM | DV | ○ | ● | ● | ● | ● | ● | ● | ● |
| CT-Wasm | [126] | Wasm | TC | ○ | ● | ○ | ● | ○ | ● | ● | ● |
| FaCT | [127] | LLVM | TC | ● | ● | ○ | ● | ○ | ● | ● | ● |
| FlowTracker | [128] | LLVM | DF | ○ | ● | ○ | ● | ○ | ● | ● | ○ |
| Jasmin | [93] | asm | DV | ○ | ● | ● | ● | ● | ● | ● | ○ |
| KMO12 | [129] | Binary | Q | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ |
| Low* | [130] | C | TC | ○ | ● | ○ | ● | ○ | ● | ● | ● |
| SC Eliminator | [131] | LLVM | DF | ● | ● | ○ | ● | ○ | ● | ● | ○ |
| Vale | [94] | asm | DF | ○ | ● | ○ | ● | ● | ● | ● | ● |
| VirtualCert | [132] | x86 | DF | ○ | ● | ○ | ● | ○ | ● | ● | ○ |

Method

TC – type-checking DF – data-flow analysis DV – deductive verification Q – Quantitative

TABLE V

OVERVIEW OF TOOLS FOR SIDE-CHANNEL RESISTANCE. SEE SECTION IV-B FOR MORE DETAILS ON TOOL FEATURES.

are devastating—key-recovery attacks have been demonstrated on real implementations, e.g., on RSA [133] and AES [134].

***How can implementation-level security fail?*** The prevailing technique for protecting against digital side-channel attacks is to follow *constant-time* coding guidelines [135]. We stress that the term is a bit of a misnomer: The idea of constant-time is that an implementation's logical execution time (not wall-clock execution time) should be independent of the values of secret data; it may, however, depend on public data, such as input length. To achieve this, constant-time implementions must follow a number of strict guidelines, e.g., they must avoid variable-time operations, control flow, and memory access patterns that depend on secret data. Unfortunately, complying with constant-time coding guidelines forces implementers to avoid natural but potentially insecure programming patterns, making enforcement error-prone.

Even worse, the observable properties of a program's execution are generally not evident from source code alone. Thus, software-invisible optimizations, e.g., compiler optimizations or data-dependent ISA optimizations, can degrade/eliminate countermeasures implemented at source-code level. Also, programmers also assume that the computing machine provides memory isolation, which is a strong and often unrealistic assumption in general-purpose hardware (e.g., due to isolation breaches allowed by speculative execution mechanisms).

***How are these failures being addressed outside CAC?*** To check that implementations correctly adhere to constant-time coding guidelines, current solutions are (1) auditing, which is costly in both time and expertise, and (2) testing, which commits the fallacy of interpreting constant-time to be constant wall-clock time. These solutions are inadequate: A botched patch for a timing vulnerability in TLS [136] led to the Lucky 13 timing vulnerability in OpenSSL [137]; in turn, the Lucky 13 patch led to yet another timing vulnerability [138]!

To prevent compiler optimizations from interfering with constant-time recipes applied at the source-code level, implementers simply avoid using compilers at all, instead choosing to implement cryptographic routines and constant-time recipes directly in assembly. Again, checking that countermeasures are implemented correctly is done through auditing and testing, but in a much more difficult, low-level setting.

Dealing with micro-architectural attacks that breach memory isolation, such as Spectre and Meltdown [139], [140], is still an open problem and seems to be out of reach of purely software-based countermeasures if there is to be any hope of achieving decent performance.

***How can computer-aided cryptography help?*** Program analysis and verification tools can automatically (or semi-automatically) check whether a given implementation meets constant-time coding guidelines, thereby providing a formal foundation supporting heretofore informal best practices. Even further, some tools can automatically repair code that violates constant-time into compliant code. Still, these approaches necessarily abstract the leakage interface available to real-world attackers. The upside is that these abstractions are precisely defined, thus clarifying the gap between between formal leakage models and real-world leakage.

***What are the fine-print caveats?*** Implementation-level proofs are only as good as their models, e.g., of physically observable effects of hardware. Furthermore, new attacks may challenge these models. Implicit assumptions arise from gaps between code and verified artifacts.

***What background do I need to know?*** Formal reasoning about side-channels is based on a *leakage model*. This model is defined over the semantics of the target language, abstractly representing what an attacker can observe during the computation process. For example, the leakage model for a branching operation may leak all values associated with the branching condition. After having defined the appropriate leakage models, proving that an implementation is secure (with respect to the leakage models) amounts to showing that the leakage accumulated over the course of execution is independent of the values of secret data. This property is an instance of *observational non-interference*.

The simplest leakage model is the program counter security model, where the program control-flow is leaked during execution [141]. The most common leakage model, namely constant-time, additionally assumes that memory accesses are leaked during execution. This leakage model is usually taken as the best practice to remove exploitable execution time variations and a best-effort against cache-attacks lauched by co-located processes.

### B. Digital Side-Channel Tools: State of the Art

Table III presents a taxonomy of tools for verifying digital side-channel resistance. Tools are listed alphabetically and are categorized as follows.

***Target (A,S).*** At what level is the analysis performed (e.g., source, assembly, binary)? To achieve the most reliable guarantees, analysis should be performed as close as possible to the executed machine code.

***Method (A).*** The tools we consider all provide a means to verify absence of timing leaks in a well-defined leakage model, but using different techniques:

- Static analysis techniques use type systems or data-flow analysis to keep track of data dependencies from secret inputs to problematic operations.
- Quantitative analysis techniques that construct a rich model of a hardware feature, e.g, the cache, and derive an upper-bound on the leaked information.
- Deductive verification techniques to prove that the leakage traces of two executions of the program coincide if the public parts of the inputs match. These techniques are closely related to the techniques used for functional correctness.

Type-checking and data-flow analysis are more amenable to automation, and they guarantee non-interference by excluding all programs that could pass secret information to an operation that appears in the trace. The emphasis on automation, however, limits the precision of the techniques, which means that secure programs may be rejected by the tools (i.e., they are not complete). Tools based deductive verification are usually complete, but require more user interaction. In some cases, users interact with the tool by annotating code, and in others the users use an interactive proof assistant to complete the proof. It is hard to conciliate a quantitative bound on leakage with standard cryptographic security notions, but such tools can also be used to prove a zero-leakage upper bound, which implies non-interference in the corresponding leakage model.

***Synthesis (U).*** Can the tool take an insecure program and automatically generate a secure program? Tools that support synthesis (e.g., FaCT [127] and SC Eliminator [131]) can automatically generate secure implementations from insecure implementations. This allows developers to write code naturally with constant-time coding recipes applied automatically.

***Soundness (A, T).*** Is the analysis sound, i.e., it only deems secure programs as secure? Note that this is our baseline filter for consideration, but we make this explicit in the table.

***Completeness (A, S).*** Is the analysis complete, i.e., it only deems insecure programs as insecure?

***Public input (S).*** Does the tool support public inputs? Support for public inputs allows differentiating between public and secret inputs. Implementations can benignly violate constant-time policies without introducing side-channel vulnerabilities by leaking no more information than public inputs of computations. Unfortunately, tools without such support would reject these implementations as insecure; forcing execution behaviors to be fully input independent may lead to large performance overheads.

***Public output (S).*** Does the tool support public outputs? Similarly, support for public outputs allows differentiating between public and secret outputs. The advantages to supporting public outputs is the same as those for supporting public inputs: for example, branching on a bit that is revealed to the attacker explicitly is fine.

***Control flow leakage (S).*** Does the tool consider control-flow leakage? The leakage model includes the list of program memory addresses accessed during program execution.

***Memory access leakage (S).*** Does the tool consider memory access pattern leakage? The leakage model includes the list of data memory addresses accessed during program execution.

***Variable-time operation leakage (S).*** Does the tool consider variable-time operation leakage? The leakage model includes the inputs to variable-time operations classified according to timing-equivalent ranges.

### C. Discussion

***Achievements: Automatic verification of constant-time real-world code.*** There are several tools that can perform verification of constant-time code automatically, both for high-level code and low-level code. These tools have been applied to real-world libraries. For example, portions of the assembly code in OpenSSL have been verified using Vale [94], high-speed implementations of SHA-3 and TLS 1.3 ciphersuites have been verified using Jasmin [93], and various off-the-shelf libraries have been analyzed with FlowTracker [128].

***Takeaway: Lowering the target provides better guarantees.*** Of the surveyed tools, several operate at the level of C code; others operate at the level of LLVM assembly; still others operate at the level of assembly or binary. The choice of target is important. To obtain a faithful correspondence with the executable program under an attacker's scrunity, analysis should be performed as close as possible to the executed machine code. Given that mainstream compilers (e.g., GCC and Clang) are known to optimize away defensive code and even introduce new side-channels [142], compiler optimizations can interfere with countermeasures deployed and verified at source-level.

***Challenge: Secure, constant-time preserving compilation.*** Given that mainstream compilers can interfere with side-channel countermeasures, many cryptography engineers avoid using compilers at all, instead choosing to implement cryptographic routines directly in assembly, which means giving up the benefits of high-level languages.

An alternative solution is to use secure compilers that carry source-level countermeasures along the compilation chain down to machine code. This way, side-channel resistant code can be written using portable C, and the secure compiler takes care of preserving side-channel resistance to specific architectures. Barthe et al. [143] laid the theoretical foundations of constant-time preserving compilation. These ideas were subsequently realized in the verified CompCert C compiler [144]. Unfortunately, CompCert-generated assembly code is not as efficient as that generated by GCC and Clang, which in turn lags the performance of hand-optimized assembly.

***Challenge: Protecting against micro-architectural attacks.*** The constant-time policy is designed to capture logical timing side channels in a simple model of hardware. Unfortunately, this simple model is inappropriate for modern hardware, as microarchitectural features, e.g., speculative or out-of-order execution, can be used for launching devastating side-channel attacks. Over the last year, the security world has been shaken by a series of attacks, including Spectre [139] and Meltdown [140]. A pressing challenge is to develop notions of constant-time security and associated verification methods that account for microarchitectural features.

***Challenge: Rethinking the hardware-software contract from secure, formal foundations.*** An instruction set architecture (ISA) describes (usually informally) what one needs to know to write a functionally correct program [145], [146]. However, current ISAs are an insufficient specification of the hardware-software contract when it comes to writing secure programs [147]. They do not capture hardware features that affect the temporal behavior of programs, which makes carrying side-channel countermeasures at the software-level to the hardware-level difficult.

To rectify this, researchers have called on new ISA designs that expose, for example, the temporal behaviors of hardware, which can lend to reasoning about them in software [147]. This, of course, poses challenging and competing requirements for hardware architects, but we believe developing formal foundations for verification and reasoning about security at the hardware-software interface can help. This line of work seems also to be the only path that can lead to a sound, formal treatment of micro-architectural attacks.

### D. Further Reading

For lack of space, we had to omit many lines of relevant work. There is a large body of work on verifying side-channel resistance in hardware [148]–[152]. There are also many tools that focus on verifying masked implementations, which aim to protect against differential power analysis attacks [153]–[158].

## V. Case Study I: Consolidating Guarantees

Previous sections focus on specific guarantees: design-level security, functional correctness, efficiency, and side-channel resistance. This case study focuses on unifying approaches that can combine these guarantees. This is a natural and important step towards the Holy Grail of computer-aided cryptography: to deliver guarantees on executable code that match the strength and elegance of guarantees on cryptographic designs.

Table VI collects implementations that verifiably meet more than one guarantee. Implementations are grouped by year (demarcated by dashed lines), starting from 2014 and ending in 2019; within each year, implementations listed alphabetically by author. We report on the primitives included, the languages targeted, the tools used, and the guarantees met.

***Computational security.*** We categorize computational security guarantees as follows: verified (●), partially verified (◐), not verified (○), and not applicable (−). The HACL$^*$-related implementations are partially verified, as only AEAD primitives have computational proofs, which are semi-mechanized [1]. Security guarantees do not apply to, e.g., elliptic curve implementations or bignum code.

***Functional correctness.*** We categorize functional correctness guarantees as follows: target-level (●), source-level (◐), and not verified (○). Target-level guarantees can be achieved in two ways: Either guarantees are established directly on assembly code, or guarantees are established at source level and a verified compiler is used.

***Efficiency.*** We categorize efficiency as follows: comparable to assembly reference implementations (●), comparable to portable C reference implementations (◐), and slower than portable C reference implementations (○).

***Side-channel resistance.*** We categorize side-channel resistance guarantees as follows: target-level (●), source-level (◐), and not verified (○).

***Takeaway: Existing tools can be used to achieve the "grand slam" of guarantees for complex cryptographic primitives.*** Ideally, we would like computational security guarantees, (target-level) functional correctness, efficiency, and (target-level) side-channel guarantees to be connected in a formal, machine-checkable way (the "grand slam" of guarantees). Many implementations come close, but so far, only one meets all four. Almeida et al. [59] formally verify an efficient implementation of the sponge construction from the SHA-3 standard. It connects proofs of RO (random oracle) indifferentiability for a pseudo-code description of the sponge construction, and proofs of functional correctness and side-channel resistance for an efficient, vectorized, implementation. The proofs are constructed using EasyCrypt and Jasmin. Other works focus on either provable security or efficiency, plus functional correctness and side-channel resistance. This disconnect is somewhat expected. Provable security guarantees are established for pseudo-code descriptions of constructions, whereas efficiency considerations demand non-trivial optimizations at the level of C or assembly.

***Takeaway: Integration can deliver strong and intuitive guarantees.*** Interpreting verification results that cover multiple requirements can be very challenging, especially because they may involve (all at once) designs, reference implementations, and optimized assembly implementations. To simplify their interpretation, Almeida et al. [161] provide a modular methodology to connect the different verification efforts, in the form of an informal meta-theorem, which concludes that an optimized assembly implementation is secure against implementation-level adversaries with side-channel capabilities. The meta-theorem states four conditions: (i) the design must be provably black-box secure in the (standard) computational model; (ii) the design is correcly implemented by a reference implementation; (iii) the reference implementation is functionally equivalent to the optimized implementation; (iv) the optimized implementation is protected against side-channels. These conditions yield a clear separation of concerns, which reflects the division of the previous sections.

***Takeaway: Achieving broad scope and efficiency.*** As Table VI illustrates, many implementations target *either* C

| Implementation(s) | | Target(s) | Tool(s) used | Computational security | Functional correctness | Efficiency | Side-channel resistance |
|---|---|---|---|---|---|---|---|
| RSA-OEAP | [159] | C | EasyCrypt, Frama-C, CompCert | ● | ● | ○ | ● |
| Curve25519 scalar mult. loop | [106] | asm | Coq, SMT | – | ● | ● | ○ |
| HMAC-SHA-2 | [160] | C | FCF, VST, CompCert | ● | ● | ○ | ○ |
| MEE-CBC | [161] | C | EasyCrypt, Frama-C, CompCert | ● | ● | ○ | ● |
| Salsa20, AES, ZUC, FFS, ECDSA, SHA-3 | [162] | Java, C | Cryptol, SAW | ○ | ◐ | ◐ | ○ |
| Curve25519 | [163] | OCaml | F*, Sage | – | ◐ | ○ | ◐ |
| Salsa20, Curve25519, Ed25519 | [93] | asm | Jasmin | ○ | ○ | ● | ● |
| IronClad (SHA-2, Poly1305, AES-CBC) | [94] | asm | Dafny, BoogieX86 | ○ | ● | ○ | ● |
| HMAC-DRBG | [164] | C | FCF, VST, CompCert | ● | ● | ○ | ○ |
| HACL*[1] | [5] | C | F* | ◐ | ◐ | ◐ | ◐ |
| HACL*[1] | [5] | C | F*, CompCert | ◐ | ● | ○ | ● |
| HMAC-DRBG | [165] | C | Cryptol, SAW | ○ | ◐ | ◐ | ○ |
| SHA-3 | [59] | asm | EasyCrypt, Jasmin | ● | ● | ● | ● |
| ChaCha20, Poly1305 | [110] | asm | EasyCrypt, Jasmin | ○ | ● | ● | ● |
| BGW multi-party computation protocol | [166] | OCaml | EasyCrypt, Why3 | ● | ◐ | ○ | ○ |
| Curve25519, P-256 | [6] | C | Fiat Crypto | – | ◐ | ◐ | ○ |
| Poly1305, AES-GCM | [95] | asm | F*, Vale | ○ | ● | ● | ● |
| Bignum code[4] | [89] | C | CryptoLine | – | ● | ◐ | ○ |
| WHACL*[1], LibSignal* | [167] | Wasm | F* | ◐ | ● | ◐ | ● |
| EverCrypt[2] | [102] | C | F* | ○ | ◐ | ◐ | ◐ |
| EverCrypt[3] | [102] | asm | F*, Vale | ○ | ● | ● | ● |

| Computational security | Functional correctness | Efficiency | Side-channel resistance |
|---|---|---|---|
| ● – verified | ● – target-level | ● – comparable to asm ref | ● – target-level |
| ◐ – partially verified | ◐ – source-level | ◐ – comparable to C ref | ◐ – source-level |
| ○ – not verified | ○ – not verified | ○ – slower than C ref | ○ – not verified |
| – – not applicable | | | |

[1](ChaCha20, Salsa20, Poly1305, SHA-2, HMAC, Curve25519, Ed25519) [2](MD5, SHA-1, SHA-2, HMAC, Poly1305, HKDF, Curve25519, ChaCha20)
[3](AES-GCM, ChaCha20, Poly1305, SHA-2, HMAC, HKDF, Curve25519, Ed25519, P-256) [4](In NaCl, wolfSSL, OpenSSL, BoringSSL, Bitcoin)

TABLE VI
VERIFIED CRYPTOGRAPHIC IMPLEMENTATIONS AND THEIR FORMAL GUARANTEES.

*or* assembly. This involves tradeoffs between the portability and relatively easy verification of C code, and the efficiency that can be gained via hand-tuned assembly. EverCrypt [102] is one of the first systems to target both. This garners the advantages of both, and it helps explain, in part, the broad scope of algorithms EverCrypt covers. Generic functionality and outer loops can be efficiently written and verified in C, whereas performance-critical cores can be verified in assembly. Soundly mixing C and assembly requires careful modeling of interoperation between the two, including platform and compiler-specific calling conventions, and differences in the "natural" memory and leakage models used to verify C versus assembly [95], [102].

## VI. Case Study II: Lessons Learned from TLS

The Transport Layer Security (TLS) protocol is widely used to establish secure channels on the Internet, and is arguably the most important real-world deployment of cryptography to date. Before TLS version 1.3, the protocol's design phases did not involve substantial academic analysis, and the process was highly reactive: When an attack was found, interim patches would be released for the mainstream TLS libraries or a longer-term fix would be incorporated in the next version of the standard. This resulted in an endless cycle of attacks and patches. Given the complexity of the protocol, early academic analyses considered only highly simplified models. However, once the academic community started considering more detailed aspects of the protocol, many new attacks were discovered, e.g., [168], [169].

The situation changed substantially during the proactive design process of TLS version 1.3: The academic community was actively consulted and encouraged to provide analysis during the process of developing multiple drafts. (See [170] for a more detailed account of TLS's standardization history.)

On the computer-aided cryptography side of things, there were substantial efforts in verifying implementations of TLS 1.3 [1], [3] and using tools to analyze symbolic [2]–[4] and computational [3] models of TLS. Below we collect the most important lessons learned from TLS throughout the years.

***Lesson: The process of formally specifying and verifying a protocol can reveal flaws.*** Prior work demonstrates that the process of formally verifying TLS, and perhaps even just formally specifying it, can reveal flaws. The implementation of TLS 1.2 with verified cryptographic security by Bhargavan et al. [60] discovered new alert fragmentation and fingerprinting attacks and led to the discovery of the Triple Handshake attacks [8]. The symbolic analysis of TLS 1.3 draft 10 using Tamarin by Cremers et al. [2] uncovered a potential attack allowing an adversary to impersonate a client during a PSK-resumption handshake, which was fixed in draft 11. The symbolic and computational analysis of TLS 1.3 draft 18 using ProVerif and CryptoVerif by Bhargavan et al. [3] uncovered a new attack on 0-RTT client authentication that was fixed in draft 13. The symbolic analysis draft 21 using Tamarin by Cremers et al. [4] revealed unexpected behavior that inhibited certain strong authentication guarantees. In nearly all cases, these discoveries led to improvements to the protocol, and otherwise clarified documentation of security guarantees.

***Lesson: Cryptographic protocol designs are moving targets; machine-checked proofs can be more easily updated.***

The TLS 1.3 specification was a rapidly moving target, with significant changes being effected on a fairly regular basis. As changes were made between a total of 28 drafts, previous analyses were often rendered stale within the space of a few months, requiring new analyses and proofs. An important benefit of machine-checked analyses and proofs over their manual counterparts is that they can be more easily updated from draft to draft as the protocol evolves [2]–[4]. Moreover, machine-checked analyses and proofs can ensure that new flaws are not introduced as components are changed.

***Lesson: Standardization processes can facilitate analysis by embracing minor changes that simplify security arguments and help modular reasoning.*** In contrast to other protocol standards, the TLS 1.3 design incorporates many suggestions from the academic community: In addition to security fixes, these include changes purposed to simplify security proofs and automated analysis. For example, this includes changes to the key schedule that help with key separation, thus simplifying modular proofs; a consistent tagging scheme; and including more transcript information in exchanges, which simplifies consistency proofs. These changes have negligible impact on the performance of the protocol, and have helped make analyzing such a complex protocol feasible.

## VII. Concluding Remarks

### A. Recommendations to Authors

Our first recommendation concerns the clarity of trust assumptions. We observe that, in some papers, the distinction between what parts of an artifact are trusted/untrusted is not always clear, which runs the risk of hazy/exaggerated guarantees. On the one hand, crisply delineating between what is trusted/untrusted may be difficult, especially when multiple tools are used, and authors may be reluctant to spell out an artifact's weaknesses. On the other hand, transparency and clarity of trust assumptions are vital for progress. We point to the paper by Beringer et al. [160] as an exemplar for how to clearly delineate between what is trusted/untrusted. At the same time, critics should understand that trust assumptions are often necessary to make progress at all.

Our second recommendation concerns the use of metrics. Metrics can be useful for measuring progress over time when used appropriately. The HACL* [5] paper uses metrics effectively: To quantify verification effort, the authors report proof-to-code ratios and person efforts for various primitives. While these are crude proxies, because the comparison is vertical (same tool, same developers), the numbers sensibly demonstrate that, e.g., code involving bignums requires more work to verify in F*. Despite their limitations, we argue that even crude metrics (when used appropriately) are better than none for advancing the field. When used inappropriately, however, metrics become dangerous and misleading. Horizontal comparisons across disparate tools tend to be problematic and must be done with care if they are to be used. For example, lines of proof or analysis times across disparate tools are often incomparable, since it is non-trivial to model a problem in the exact same way.

### B. Recommendations to Tool Developers

Although we are still in the early days of seeing verified cryptography deployed in the wild, one major pending challenge is how to make computer-aided cryptography artifacts maintainable. Because computer-aided cryptography tools sit at the bleeding-edge of how cryptography is done, they are constantly evolving, often in non-backwards-compatible ways. When this happens, we must either leave many artifacts (e.g., machine-checked proofs) to become stale, or else muster significant human efforts to keep them up to date. Moreover, because cryptography is a moving target, we should expect that even verified implementations (and their proofs) will require updates. This could be to add additional functionality, or worse, to swiftly patch new vulnerabilities beyond what was verifiably accounted for. If only a handful of experts are capable of maintenance, then, in this respect, we are in no better position than we are today. We hope to see more interplay between proof engineering research [171], [172] and computer-aided cryptography research in the coming years.

### C. Recommendations to Standardization Bodies

Given its benefits in the TLS 1.3 standardisation effort, we believe computer-aided cryptography should play an important role in the cryptography standardization process [173]. Traditionally, cryptographic standards are written in a combination of prose, formulas, and pseudocode, and can change drastically from draft to draft. On top of getting the cryptography right in the first place, standards must also focus on clarity, ease of implementation, and interoperability. It is perhaps not surprising, then, that the standardization process can be long and arduous. And even when it is successful, the substantial gap between standards and implementations still leaves plenty of rope for error.

Security proofs can also become a double-edged sword in standardization processes. Proposals supported by handwritten security arguments often cannot be reasonably audited. A plausible claim with a proof that cannot be audited should not be taken as higher assurance than simply stating the claim—we argue that the latter is a lesser evil, as it does not create a false sense of security. As a concrete example, Hales [174] discusses ill-intentioned security arguments in the context of the Dual EC pseudo-random generator [175]. Another example is the recent discovery of attacks against the AES-OCB2 ISO standard, which was previously believed to be provably secure [176].

To address these challenges, we advocate the use of computer-aided cryptography, not only to formally certify compliance to standards, but also to facilitate the role of auditors and evaluators in standardization processes, allowing the discussion to focus on the security claims, rather than on whether the supporting security arguments are convincing. We see the current NIST post-quantum standardization effort [177] as an excellent opportunity to put our recommendations into practice, and we encourage the computer-aided cryptography community to engage in the process.

# References


[1] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2017, pp. 463–482.

[2] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-rtt, resumption and delayed authentication," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2016, pp. 470–485.

[3] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2017, pp. 483–502.

[4] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1773–1788.

[5] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1789–1806.

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

[7] N. Drucker and S. Gueron, "Selfie: reflections on TLS 1.3 with PSK," Cryptology ePrint Archive, Report 2019/347, 2019, https://eprint.iacr.org/2019/347.

[8] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2014, pp. 98–113.

[9] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 4004. Springer, 2006, pp. 409–426.

[10] S. Halevi, "A plausible approach to computer-aided cryptographic proofs," *IACR Cryptology ePrint Archive*, vol. 2005, p. 181, 2005.

[11] K. G. Paterson and G. J. Watson, "Plaintext-dependent decryption: A formal security treatment of SSH-CTR," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 6110. Springer, 2010, pp. 345–361.

[12] A. Boldyreva, J. P. Degabriele, K. G. Paterson, and M. Stam, "Security of symmetric encryption in the presence of ciphertext fragmentation," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 7237. Springer, 2012, pp. 682–699.

[13] J. P. Degabriele, K. G. Paterson, and G. J. Watson, "Provable security in the real world," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 33–41, 2011.

[14] Y. Lindell, "How to simulate it - A tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography.* Springer International Publishing, 2017, pp. 277–346.

[15] S. F. Doghmi, J. D. Guttman, and F. J. Thayer, "Searching for shapes in cryptographic protocols," in *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 4424. Springer, 2007, pp. 523–537.

[16] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, pp. 8:1–8:45, 2011.

[17] M. Backes, C. Hriţcu, and M. Maffei, "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations," *J. Comput. Secur.*, vol. 22, no. 2, pp. 301–353, Mar. 2014.

[18] S. Escobar, C. A. Meadows, and J. Meseguer, "Maude-npa: Cryptographic protocol analysis modulo equational properties," in *Foundations of Security Analysis and Design (FOSAD)*, ser. Lecture Notes in Computer Science, vol. 5705. Springer, 2007, pp. 1–50.

[19] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth, "ProVerif: Cryptographic protocol verifier in the formal model," 2010.

[20] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 1, 2008.

[21] M. Arapinis, E. Ritter, and M. D. Ryan, "Statverif: Verification of stateful processes," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2011, pp. 33–47.

[22] C. J. F. Cremers, "The scyther tool: Verification, falsification, and analysis of security protocols," in *Proc. of the International Conference on Computer-Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 5123. Springer, 2008, pp. 414–418.

[23] S. Meier, C. J. F. Cremers, and D. A. Basin, "Strong invariants for the efficient construction of machine-checked protocol security proofs," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2010, pp. 231–245.

[24] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *Proc. of the International Conference on Computer-Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013, pp. 696–701.

[25] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2014, pp. 163–178.

[26] M. Turuani, "The cl-atse protocol analyser," in *Proc. of the International Conference on Term Rewriting and Applications (RTA)*, ser. Lecture Notes in Computer Science, vol. 4098. Springer, 2006, pp. 277–286.

[27] D. A. Basin, S. Mödersheim, and L. Viganò, "OFMC: A symbolic model checker for security protocols," *Int. J. Inf. Sec.*, vol. 4, no. 3, pp. 181–208, 2005.

[28] A. Armando and L. Compagna, "SATMC: A sat-based model checker for security protocols," in *Proc. of the European Conference on Logics in Artificial Intelligence (JELIA)*, ser. Lecture Notes in Computer Science, vol. 3229. Springer, 2004, pp. 730–733.

[29] R. Chadha, V. Cheval, Ştefan Ciobâcă, and S. Kremer, "Automated verification of equivalence properties of cryptographic protocols," *ACM Trans. Comput. Log.*, vol. 17, no. 4, pp. 23:1–23:32, 2016.

[30] V. Cheval, "APTE: an algorithm for proving trace equivalence," in *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 8413. Springer, 2014, pp. 587–592.

[31] V. Cheval, S. Kremer, and I. Rakotonirina, "DEEPSEC: deciding equivalence properties in security protocols theory and practice," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2018, pp. 529–546.

[32] V. Cortier, A. Dallon, and S. Delaune, "Sat-equiv: An efficient tool for equivalence properties," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2017, pp. 481–494.

[33] A. Tiu and J. E. Dawson, "Automating open bisimulation checking for the spi calculus," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2010, pp. 307–321.

[34] J. K. Millen, "A necessarily parallel attack," in *In Workshop on Formal Methods and Security Protocols*, 1999.

[35] N. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov, "Multiset rewriting and the complexity of bounded security protocols," *Journal of Computer Security*, vol. 12, no. 2, pp. 247–311, 2004.

[36] J. Dreier, C. Duménil, S. Kremer, and R. Sasse, "Beyond subterm-convergent equational theories in automated verification of stateful protocols," in *Proc. of the International Conference on Principles of Security and Trust (POST)*. Springer-Verlag, 2017.

[37] C. Cremers and D. Jackson, "Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 78–93.

[38] S. Delaune and L. Hirschi, "A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols," *J. Log. Algebr. Meth. Program.*, vol. 87, pp. 127–144, 2017.

[39] L. C. Paulson, *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, ser. Lecture Notes in Computer Science. Springer, 1994, vol. 828.

[40] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Verified

cryptographic implementations for TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 3:1–3:32, 2012.

[41] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 435–450.

[42] N. Kobeissi, G. Nicolas, and K. Bhargavan, "Noise explorer: Fully automated modeling and verification for arbitrary noise protocols," in *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 356–370.

[43] D. A. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 1383–1396.

[44] C. Cremers, M. Dehnel-Wild, and K. Milner, "Secure authentication in the grid: A formal analysis of DNP3 SAv5," *Journal of Computer Security*, vol. 27, no. 2, pp. 203–232, 2019.

[45] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," *Journal of Logic and Algebraic Programming*, vol. 75, no. 1, pp. 3–51, Feb.–Mar. 2008.

[46] 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 (STM)*, ser. Lecture Notes in Computer Science, vol. 8743. Berlin, Heidelberg: Springer, Sep. 2014, pp. 162–177.

[47] D. Basin, J. Dreier, and R. Casse, "Automated symbolic proofs of observational equivalence," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. New York, NY: ACM Press, Oct. 2015, pp. 1144–1155.

[48] G. Barthe, B. Grégoire, and B. Schmidt, "Automated proofs of pairing-based cryptography," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2015, pp. 1156–1168.

[49] G. Barthe, B. Grégoire, and S. Z. Béguelin, "Formal certification of code-based cryptographic proofs," in *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2009, pp. 90–101.

[50] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "Crypthol: Game-based proofs in higher-order logic," *IACR Cryptology ePrint Archive*, vol. 2017, p. 753, 2017.

[51] B. Blanchet, "CryptoVerif: Computationally sound mechanized prover for cryptographic protocols," in *Dagstuhl seminar on Formal Protocol Verification Applied*, 2007, p. 117.

[52] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Proc. of the Annual Cryptology Conference (CRYPTO)*, ser. Lecture Notes in Computer Science, vol. 6841. Springer, 2011, pp. 71–90.

[53] 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 *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 256–270.

[54] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *Proc. of the International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 9036. Springer, 2015, pp. 53–72.

[55] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Z. Béguelin, "Fully automated analysis of padding-based encryption in the computational model," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 1247–1260.

[56] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, ser. CCS '11, 2011, pp. 341–350.

[57] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the wireguard virtual private network protocol," in *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 231–246.

[58] J. B. Almeida, M. Barbosa, G. Barthe, M. Campagna, E. Cohen, B. Grégoire, V. Pereira, B. Portela, P. Strub, and S. Tasiran, "A machine-checked proof of security for AWS key management service," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 63–78.

[59] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P. Strub, "Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 1607–1622.

[60] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2013, pp. 445–459.

[61] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin, "Proving the tls handshake secure (as it is)," in *Proc. of the Annual Cryptology Conference (CRYPTO)*, 2014.

[62] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2001, pp. 104–115.

[63] 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 *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2015, pp. 535–552.

[64] C. E. Landwehr, D. Boneh, J. C. Mitchell, S. M. Bellovin, S. Landau, and M. E. Lesk, "Privacy and cybersecurity: The next 100 years," *Proceedings of the IEEE*, vol. 100, no. Centennial-Issue, pp. 1659–1673, 2012.

[65] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proc. of the IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 2001, pp. 136–145.

[66] K. Liao, M. A. Hammer, and A. Miller, "ILC: a calculus for composable, computational cryptography," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019, pp. 640–654.

[67] R. Canetti, A. Stoughton, and M. Varia, "Easyuc: Using easycrypt to mechanize proofs of universally composable security," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 167–183.

[68] A. Lochbihler, S. R. Sefidgar, D. A. Basin, and U. Maurer, "Formalizing constructive cryptography using crypthol," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 152–166.

[69] J. A. Akinyele, M. Green, and S. Hohenberger, "Using SMT solvers to automate design tasks for encryption and signature schemes," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013, pp. 399–410.

[70] A. J. Malozemoff, J. Katz, and M. D. Green, "Automated analysis and synthesis of block-cipher modes of operation," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2014, pp. 140–152.

[71] V. T. Hoang, J. Katz, and A. J. Malozemoff, "Automated analysis and synthesis of authenticated encryption schemes," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2015, pp. 84–95.

[72] G. Barthe, E. Fagerholm, D. Fiore, A. Scedrov, B. Schmidt, and M. Tibouchi, "Strongly-optimal structure preserving signatures from type II pairings: synthesis and lower bounds," *IET Information Security*, vol. 10, no. 6, pp. 358–371, 2016.

[73] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: nearly practical verifiable computation," *Commun. ACM*, vol. 59, no. 2, pp. 103–112, 2016.

[74] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 253–270.

[75] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *Proc. of USENIX Security*, 2012.

[76] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proc. of CRYPTO*, 2013.

[77] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A. Sadeghi, and T. Schneider, "A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols," in *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pp. 151–167.

[78] M. Fredrikson and B. Livshits, "Zø: An optimizing distributing zero-knowledge compiler," in *Proc. of the USENIX Security Symposium (USENIX)*, 2014, pp. 909–924.

[79] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, "ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash," in *Proc. of USENIX Security*, 2010.

[80] M. Backes, M. Maffe, and K. Pecina, "Automated synthesis of privacy-preserving distributed applications," in *Proc. of ISOC NDSS*, 2012.

[81] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1220–1237.

[82] J. B. Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Z. Béguelin, "Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 488–500.

[83] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, and V. Pereira, "A fast and verified software stack for secure function evaluation," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1989–2006.

[84] C. Fournet, C. Keller, and V. Laporte, "A certified compiler for verifiable computing," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*, 2016, pp. 268–280.

[85] A. Rastogi, N. Swamy, and M. Hicks, "Wys*: A DSL for verified secure multi-party computations," in *Proc. of the International Conference on Principles of Security and Trust (POST)*, 2019, pp. 99–122.

[86] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *Proc. of the International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 7215. Springer, 2012, pp. 3–29.

[87] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *J. Autom. Reasoning*, vol. 46, no. 3-4, pp. 225–259, 2011.

[88] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb, "Constructing semantic models of programs with the software analysis workbench," in *Proc. of the International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE)*, ser. Lecture Notes in Computer Science, vol. 9971, 2016, pp. 56–72.

[89] Y. Fu, J. Liu, X. Shi, M. Tsai, B. Wang, and B. Yang, "Signed cryptographic program verification with typed cryptoline," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 1591–1606.

[90] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proc. of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, ser. Lecture Notes in Computer Science, vol. 6355. Springer, 2010, pp. 348–370.

[91] A. Polyakov, M. Tsai, B. Wang, and B. Yang, "Verifying arithmetic assembly programs in cryptographic primitives (invited talk)," in *Proc. of the International Conference on Concurrency Theory (CONCUR)*, ser. LIPIcs, vol. 118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 4:1–4:16.

[92] D. J. Bernstein and P. Schwabe, "gfverif: Fast and easy verification of finite-field arithmetic," 2016. [Online]. Available: http://gfverif.cryptojedi.org

[93] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1807–1823.

[94] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2017, pp. 917–934.

[95] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, "A verified, efficient embedding of a verifiable assembly language," *PACMPL*, vol. 3, no. POPL, pp. 63:1–63:30, 2019.

[96] A. W. Appel, "Verified software toolchain - (invited talk)," in *Proc. of the European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 6602. Springer, 2011, pp. 1–17.

[97] J. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *Proc. of the European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 7792. Springer, 2013, pp. 125–128.

[98] D. J. Bernstein and T. Lange, "ebacs: Ecrypt benchmarking of cryptographic systems," 2009.

[99] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of heartbleed," in *Proc. of the Internet Measurement Conference (IMC)*. ACM, 2014, pp. 475–488.

[100] S. Gueron and V. Krasnov, "The fragility of AES-GCM authentication algorithm," in *Proceedings of the Conference on Information Technology: New Generations*, Apr. 2014.

[101] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical realisation and elimination of an ecc-related software bug attack," in *Proc. of the Cryptographers' Track at the RSA Conference (CT-RSA)*, ser. Lecture Notes in Computer Science, vol. 7178. Springer, 2012, pp. 171–186.

[102] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Z. Béguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," *IACR Cryptology ePrint Archive*, vol. 2019, p. 757, 2019.

[103] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[104] T. Oliveira, J. L. Hernandez, H. Hisil, A. Faz-Hernández, and F. Rodríguez-Henríquez, "How to (pre-)compute a ladder - improving the performance of X25519 and X448," in *Proc. of the International Conference on Selected Areas in Cryptography (SAC)*, ser. Lecture Notes in Computer Science, vol. 10719. Springer, 2017, pp. 172–191.

[105] T. Chou, "Sandy2x: New curve25519 speed records," in *Proc. of the International Conference on Selected Areas in Cryptography (SAC)*, ser. Lecture Notes in Computer Science, vol. 9566. Springer, 2015, pp. 145–160.

[106] Y. Chen, C. Hsu, H. Lin, P. Schwabe, M. Tsai, B. Wang, B. Yang, and S. Yang, "Verifying curve25519 software," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 299–309.

[107] "curve25519-donna: Implementations of a fast Elliptic-curve Diffie-Hellman primitive," https://github.com/agl/curve25519-donna.

[108] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill, "Ironfleet: proving practical distributed systems correct," in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 1–17.

[109] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Proc. of the IACR International Conference on Practice and Theory of Public-Key Cryptography (PKC)*, ser. Lecture Notes in Computer Science, vol. 3958. Springer, 2006, pp. 207–228.

[110] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," *CoRR*, vol. abs/1904.04606, 2019.

[111] J. P. Lim and S. Nagarakatte, "Automatic equivalence checking for assembly implementations of cryptography libraries," in *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization, (CGO)*. IEEE, 2019, pp. 37–49.

[112] "The GNU Multiple Precision Arithmetic Library." [Online]. Available: https://gmplib.org/

[113] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, 2014.

[114] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," in *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2015, pp. 595–608.

[115] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying security invariants in ExpressOS," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013, pp. 293–304.

[116] G. Morrisett, G. Tan, J. Tassarotti, J. Tristan, and E. Gan, "Rocksalt: better, faster, stronger SFI for the x86," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 395–404.

[117] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, “Jitk: A trustworthy in-kernel interpreter infrastructure,” in *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014, pp. 33–47.

[118] 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 *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 18–37.

[119] A. Vasudevan, S. Chaki, L. Jia, J. M. McCune, J. Newsome, and A. Datta, “Design, implementation and verification of an extensible and modular hypervisor framework,” in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2013, pp. 430–444.

[120] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, “Verdi: a framework for implementing and formally verifying distributed systems,” in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2015, pp. 357–368.

[121] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016, pp. 614–630.

[122] 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 *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014, pp. 165–181.

[123] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, “Formal verification of side-channel countermeasures using self-composition,” *Sci. Comput. Program.*, vol. 78, no. 7, pp. 796–812, 2013.

[124] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, “Cacheaudit: A tool for the static analysis of cache side channels,” in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2013, pp. 431–446.

[125] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2016, pp. 53–70.

[126] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, “Ct-wasm: type-driven secure cryptography for the web ecosystem,” *PACMPL*, vol. 3, no. POPL, pp. 77:1–77:29, 2019.

[127] 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 *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019, pp. 174–189.

[128] B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha, “Sparse representation of implicit flows with applications to side-channel detection,” in *Proc. of the International Conference on Compiler Construction (CC)*. ACM, 2016, pp. 110–120.

[129] B. Köpf, L. Mauborgne, and M. Ochoa, “Automatic quantification of cache side-channels,” in *Proc. of the International Conference on Computer-Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 7358. Springer, 2012, pp. 564–580.

[130] J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, “Verified low-level programming embedded in F,” *PACMPL*, vol. 1, no. ICFP, pp. 17:1–17:29, 2017.

[131] M. Wu, S. Guo, P. Schaumont, and C. Wang, “Eliminating timing side-channel leaks using program repair,” in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 15–26.

[132] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie, “System-level non-interference for constant-time cryptography,” in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 1267–1279.

[133] D. Brumley and D. Boneh, “Remote timing attacks are practical,” in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2003.

[134] D. J. Bernstein, “Cache-timing attacks on AES,” 2005.

[135] J.-P. Aumasson, “Guidelines for Low-Level Cryptography Software,” https://github.com/veorq/cryptocoding.

[136] B. Moller, “Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures,” 2004. [Online]. Available: http://www.openssl.org/~bodo/tls-cbc.txt

[137] N. J. AlFardan and K. G. Paterson, “Lucky thirteen: Breaking the TLS and DTLS record protocols,” in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2013, pp. 526–540.

[138] J. Somorovsky, “Curious padding oracle in OpenSSL (cve-2016-2107),” 2016. [Online]. Available: https://web-in-security.blogspot.com/2016/05/curious-padding-oracle-in-openssl-cve.html

[139] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 1–19.

[140] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2018, pp. 973–990.

[141] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, “The program counter security model: Automatic detection and removal of control-flow side channel attacks,” in *Proc. of the International Conference on Information Security and Cryptology (ICISC)*, ser. Lecture Notes in Computer Science, vol. 3935. Springer, 2005, pp. 156–168.

[142] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, “When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015,” in *Proc. of the International Conference on Cryptology and Network Security (CANS)*, ser. Lecture Notes in Computer Science, vol. 10052, 2016, pp. 573–582.

[143] G. Barthe, B. Grégoire, and V. Laporte, “Secure compilation of side-channel countermeasures: The case of cryptographic ”constant-time”,” in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2018, pp. 328–343.

[144] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, “Formal verification of a constant-time preserving c compiler,” Cryptology ePrint Archive, Report 2019/926, 2019, https://eprint.iacr.org/2019/926.

[145] A. Reid, “Trustworthy specifications of arm® v8-a and v8-m system level architecture,” in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. IEEE, 2016, pp. 161–168.

[146] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, “ISA semantics for armv8-a, risc-v, and CHERI-MIPS,” *PACMPL*, vol. 3, no. POPL, pp. 71:1–71:31, 2019.

[147] G. Heiser, “For safety’s sake: We need a new hardware-software contract!” *IEEE Design & Test*, vol. 35, no. 2, pp. 27–30, 2018.

[148] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, “A hardware design language for timing-sensitive information-flow security,” in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015, pp. 503–516.

[149] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009, pp. 109–120.

[150] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, “Sapper: a language for hardware-level security policy enforcement,” in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014, pp. 97–112.

[151] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, “Caisson: a hardware description language for secure information flow,” in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 109–120.

[152] K. von Gleissenthall, R. G. Kici, D. Stefan, and R. Jhala, “IODINE: verifying constant-time execution of hardware,” in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2019, pp. 1411–1428.

[153] H. Eldib, C. Wang, and P. Schaumont, "Smt-based verification of software countermeasures against side-channel attacks," in *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 8413. Springer, 2014, pp. 62–77.

[154] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Proc. of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, vol. 8086. Springer, 2013, pp. 293–310.

[155] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Proc. of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, vol. 7428. Springer, 2012, pp. 58–75.

[156] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *Proc. of the International Conference on Computer-Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 114–130.

[157] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub, "Verified proofs of higher-order masking," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 9056. Springer, 2015, pp. 457–485.

[158] G. Barthe, S. Belaïd, G. Cassiers, P. Fouque, B. Grégoire, and F. Standaert, "maskverif: Automated verification of higher-order masking in presence of physical defaults," in *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, ser. Lecture Notes in Computer Science, vol. 11735. Springer, 2019, pp. 300–318.

[159] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 1217–1230.

[160] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of openssl HMAC," in *Proc. of the USENIX Security Symposium (USENIX)*. USENIX Association, 2015, pp. 207–221.

[161] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC," in *Proc. of the International Conference on Fast Software Encryption (FSE)*, ser. Lecture Notes in Computer Science, vol. 9783. Springer, 2016, pp. 163–184.

[162] A. Tomb, "Automated verification of real-world cryptographic implementations," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 26–33, 2016.

[163] J. K. Zinzindohoue, E. Bartzia, and K. Bhargavan, "A verified extensible library of elliptic curves," in *Proc. of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2016, pp. 296–309.

[164] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, "Verified correctness and security of mbedtls HMAC-DRBG," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 2007–2020.

[165] 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 *Proc. of the International Conference on Computer-Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 10982. Springer, 2018, pp. 430–446.

[166] K. Eldefrawy and V. Pereira, "A high-assurance evaluator for machine-checked secure multiparty computation," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 851–868.

[167] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in webassembly," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 1256–1274.

[168] C. Meyer and J. Schwenk, "Sok: Lessons learned from SSL/TLS attacks," in *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 8267. Springer, 2013, pp. 189–209.

[169] J. Clark and P. C. van Oorschot, "Sok: SSL and HTTPS: revisiting past challenges and evaluating certificate trust model enhancements," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2013, pp. 511–525.

[170] K. G. Paterson and T. van der Merwe, "Reactive and proactive standardisation of TLS," in *Proc. of the International Conference on Security Standardisation Research (SSR)*, ser. Lecture Notes in Computer Science, vol. 10074. Springer, 2016, pp. 160–186.

[171] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, "QED at large: A survey of engineering of formally verified software," *Foundations and Trends in Programming Languages*, vol. 5, no. 2-3, pp. 102–281, 2019.

[172] D. R. Jeffery, M. Staples, J. Andronick, G. Klein, and T. C. Murray, "An empirical research agenda for understanding formal methods productivity," *Information & Software Technology*, vol. 60, pp. 102–112, 2015.

[173] K. Bhargavan, F. Kiefer, and P. Strub, "hacspec: Towards verifiable crypto standards," in *Proc. of the International Conference on Security Standardisation Research (SSR)*, ser. Lecture Notes in Computer Science, vol. 11322. Springer, 2018, pp. 1–20.

[174] T. C. Hales, "The nsa back door to nist," *Notices of the AMS*, vol. 61, no. 2, pp. 190–19.

[175] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the juniper dual EC incident," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016, pp. 468–479.

[176] A. Inoue, T. Iwata, K. Minematsu, and B. Poettering, "Cryptanalysis of OCB2: attacks on authenticity and confidentiality," in *Proc. of the Annual Cryptology Conference (CRYPTO)*, 2019, pp. 3–31.

[177] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on post-quantum cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.