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, [‖]MIT, [**]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, scope, 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 widely deployed 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 make up primitives, primitives make up protocols, and protocols make up 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 "cores" of cryptographic mechanisms to simplify analysis, yet remain highly complex and error-prone; demands for aggressively optimized implementations greatly increase the risks of introducing bugs, which are difficult to catch by code testing or auditing; ad-hoc constant-time coding recipes for mitigating side-channel attacks are tricky to implement, and yet may not cover the 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. It 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, even revealing 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 design specifications on all possible inputs. At the deployment level, tools can check that implementations correctly protect against 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 were influential in the recent standardization of TLS 1.3 [1]–[4]. Formally verified code is also being deployed at Internet-scale—components of the HACL* library [5] are being integrated into Mozilla Firefox's NSS security engine, elliptic curve code generated using the Fiat Cryptography framework [6] has populated Google's BoringSSL library, and EverCrypt [7] routines are used in the Zinc crypto library for the Linux kernel. In light of these successes, there is growing enthusiasm for computer-aided cryptography. This is reflected in the rapid emergence of a dynamic community comprised of theoretical and applied cryptographers, cryptography engineers, and formal methods practitioners. Together, the community aims to achieve broader adoption of computer-aided cryptography, blending ideas from many fields, and more generally, to contribute to the future development of cryptography.

At the same time, computer-aided cryptography risks becoming a victim of its own success. Trust in the field can be undermined by difficulties in understanding the guarantees and fine-print caveats of computer-aided cryptography artifacts. The field is also 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 lower-level primitives and protocols to whole 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 and open source projects) 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 section with a critical review of the area, 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 state-of-the-art tools based on criteria along four main categories: *accuracy (A)*, *scope (S)*, *trust (T)*, and *usability (U)*. For each criterion, we label them with one or more categories, explain their importance, and provide some light discussion about tool support for them. 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 cover, we are unable to be exhaustive in each area, but we still point to other relevant lines of work.

Sections V and VI describe two case studies. Our first case study (Section V) examines how to combine tools that address different parts of the problem space and consolidate their guarantees. Our second case study (Section VI) distills the lessons learned from the computer-aided cryptography community's involvement in the TLS 1.3 standardization effort.

Finally, in Section VII, we offer recommendations to paper authors, tool developers, and standardization bodies on how to best 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 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 their security against entire classes of attacks. This has become standard practice in cryptography, and security proofs are necessary for any new standard. This holds true at all levels: primitives, protocols, and systems. When using a lower-level component in a larger system, it is crucial to understand what security notion and adversarial model the proof is relative to. Similar considerations apply when evaluating the security of a cryptographic system relative to its intended deployment environment.

How can design-level security fail? The current modus operandi for validating the security of 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, attacks are often found outside of this core.

How are these failures being addressed outside CAC? To minimize erroneous arguments, cryptographers have devised a number of methodological frameworks for security analysis (e.g., the code-based game playing [8] and universal composability [9] frameworks). The high-level goal of these frameworks is to decompose security arguments into simpler arguments that are easier to get right and then smoothly combine the results. Still, pen-and-paper proofs based on these methodologies remain complex and 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, complicating 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 crystallize the benefits of on-paper methodologies and of real-world provable security. They also deliver trustworthy analyses for complex designs that are beyond reach of pen-and-paper analysis.

What are the fine-print caveats? Computer-aided security proofs are only as good as the statements being proven. However, understanding these statements can be challenging. Most security proofs rely on implicit assumptions; 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 with any software, tools may have bugs.

What background do I need to know about symbolic security? The symbolic model is an abstract model for representing and analyzing cryptographic protocols. Messages (e.g., keys, nonces) are represented symbolically as *terms* (in the parlance of formal logic). Typically, terms are atomic data, meaning that they cannot be split into, say, component bitstrings. Cryptographic primitives are modeled as black-box functions over terms related by a set of mathematical identities called an *equational theory*. For example, symmetric encryption can be modeled by the black-box functions Enc and Dec related by the following equational theory: $\mathsf{Dec}(\mathsf{Enc}(m, k), k) = m$. This says that decrypting the ciphertext $\mathsf{Enc}(m, k)$ using the key k recovers the original plaintext m.

An adversary is restricted to compute (i.e., derive new terms contributing to its *knowledge set*) using only the specified primitives and equational theory. Equational theories are thus important for broadening the scope of analysis—ignoring valid equations implicitly weakens the class of adversaries considered. In the example above, m and k are atomic terms, and so equipped with only the given identity, an adversary can decrypt a ciphertext only if it has knowledge of the *entire* secret key. Such simplifications enable modeling and verifying protocols using symbolic logic. Symbolic tools are thus well-suited to automatically searching for and unveiling logical flaws in complex cryptographic protocols and systems.

Symbolic security properties come in two main flavors: trace properties and equivalence properties. Trace properties state that a bad event never occurs on any execution trace. For example, a protocol preserves trace-based secrecy if, for any execution trace, secret data is not in the adversarial knowledge set. On the other hand, equivalence properties state that an adversary is unable to distinguish between two protocols, often with one being the security specification. Equivalence properties typically cannot be (naturally or precisely) expressed as trace properties. For example, a protocol preserves indistinguishability-based secrecy if the adversary cannot differentiate between a trace with the real secret and a trace with the real secret replaced by a random value.

What background do I need to know about computational security? In the computational model, messages are bitstrings, cryptographic primitives are probabilistic algorithms on bitstrings, and adversaries are probabilistic Turing machines. For example, symmetric encryption can be modeled by a triple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. The probabilistic key generation algorithm Gen outputs a bitstring k. The encryption (decryption) algorithm Enc (Dec) takes as input a key k and a plaintext m (ciphertext c), and outputs a ciphertext c (plaintext m). The basic correctness property that must hold for every key k output by Gen and every message m in the message space is $\mathsf{Dec}(\mathsf{Enc}(m, k), k) = m$. Because keys are bitstrings in this model, knowing bits of an encryption key reduces the computational resources required to decrypt a ciphertext.

Computational security properties are also probabilistic and can be characterized along two axes: game-based or simulation-based, and concrete or asymptotic.

Game-based properties specify a probabilistic experiment called a "game" between a challenger and an adversary, and an explicit goal condition that the adversary must achieve to break a scheme. Informally, security statements say: For all adversaries, the probability of achieving the goal condition does not exceed some threshold. The specific details, e.g., the adversary's computational resources and the threshold, depend on the choice of concrete or asymptotic security.

A core proof methodology for game-based security is *game hopping*. In the originally specified game, the adversary's success probability may be unknown. Thus, we proceed by step-wise transforming the game until reaching one in which the success probability can be computed. We also bound the increases in the success probability from the game transformations, often by reducing to an assumed hard problem (e.g., the discrete log or RSA problems). We can then deduce a bound on the adversary's success probability in the original game. The interested reader can see the tutorials on game hopping by Shoup [14] and Bellare and Rogaway [8].

Simulation-based properties specify two probabilistic experiments: The "real" game runs the scheme under analysis. The "ideal" game runs an idealized scheme that does not involve any cryptography, but instead runs a trusted third-party called an *ideal functionality*, which serves as the security specification. Informally, security statements say: For all adversaries in the real game, there exists a *simulator* in the ideal game that can translate any attack on the real scheme into an attack on the ideal functionality. Because the ideal functionality is secure by definition, the real scheme must also be secure. In general, simulation-based proofs tend to be more complicated than game-based proofs, but importantly they support composition theorems that allow analyzing complex constructions in a modular way from simpler building blocks. The interested reader can see the tutorial on simulation-based proofs by Lindell [15].

Concrete security quantifies the security of a scheme by bounding the maximum success probability of an adversary given some upper bound on running time. A scheme is (t, ϵ)-secure if every adversary running for time at most t succeeds in breaking the scheme with probability at most ϵ. In contrast, asymptotic security views the running time of the adversary and its success probability as functions of some security parameter (e.g., key length), rather than as concrete numbers. A scheme is secure if every probabilistic polynomial time adversary succeeds in breaking the scheme with negligible probability (i.e., with probability asymptotically less than all inverse polynomials in the security parameter).

Of these different security properties, we note that computer-aided security proofs have primarily focused on game-based, concrete security. Work on mechanizing simulation-based proofs is relatively nascent; asymptotic security is the prevailing paradigm in cryptography, but by proving concrete security, asymptotic security follows *a fortiori*.

Tool		Unbound	Trace	Equiv	Eq-thy	State	Link
CPSA$^{\triangleright}$	[16]	●	●	○	○	●	○
F7$^{\diamond}$	[17]	●	●	○	◐	●	●
↳F5$^{\diamond}$	[18]	●	●	○	◐	●	●
Maude-NPA$^{\triangleright}$	[19]	●	●	●^{d}	●	○	○
ProVerif*†	[20]	●	●	●^{d}	◐	○	○
↳fs2pv$^{\diamond\dagger}$	[21]	●	●	○	◐	○	●
↳GSVerif*†	[22]	●	●	○	◐	●	○
↳ProVerif-ATP*†	[23]	●	●	○	●	○	○
↳StatVerif*†	[24]	●	●	●^{d}	◐	●	○
Scyther$^{\triangleright}$	[25]	●	●	○	○	○	○
scyther-proof$^{\triangleright\ddagger\S}$	[26]	●	●	○	○	○	○
Tamarin*‡	[27]	●	●	●^{d}	●	●	○
↳SAPIC*	[28]	●	●	○	●	●	○
CI-AtSe$^{\triangleright}$	[29]	○	●	○	●	●	○
OFMC$^{\triangleright\dagger}$	[30]	○	●	○	◐	●	○
SATMC$^{\triangleright}$	[31]	○	●	○	○	●	○
AKISS*	[32]	○	○	●^{t}	●	●	○
APTE*	[33]	○	○	●^{t}	○	●	○
DEEPSEC*	[34]	○	○	●^{t}	◐	●	○
SAT-Equiv*	[35]	○	○	●^{t}	○	○	○
SPEC*,§	[36]	○	○	●^{o}	○	○	○

Specification language	Miscellaneous symbols
$\triangleright$ – security protocol notation	↳ – previous tool extension
$\star$ – process calculus	† – 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	o – open bisimilarity
○ – fixed	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, annotated with the relevant criteria (A, S, T, U) described in the introduction. Note that the capabilities of symbolic tools are more nuanced than what is reflected in the table—the set of examples that tools can handle varies even if they support the same features according to the table.

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 [37]. Bounded tools (○) explicitly limit the analyzed number of sessions and do not consider attacks beyond the cut-off. Unbounded tools (●) can prove the absence of attacks within the model, but at the cost of undecidability [38].

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. This is because bounded tools are a bit naive in their exploration of the state space, basically enumerating options (but exploiting some symmetry). They therefore typically grow exponentially in the number of sessions. The unbounded tools inherently need to be "more clever" to even achieve unbounded analysis. While their algorithms are more complex, when they work (i.e., terminate), the analysis is independent of the number of sessions.

Trace properties (S). Does the tool support verification of trace properties?

Equivalence properties (S). Does the tool support verification of equivalence properties? There are several different equivalence notions used in current tools. Here, we provide some high-level intuition, but for a more formal treatment, see the survey by Delaune and Hirschi [39].

Trace equivalence (t) means that, for each trace of one protocol, there exists a corresponding trace of the other protocol, such that the messages exchanged in these two traces are indistinguishable. This is the weakest equivalence notion, roughly meaning that it can express the most security properties. (The other stronger notions are often intermediate steps towards proving trace equivalence.) It is also arguably the most natural for formalizing privacy properties.

Open bisimilarity (o) is a strictly stronger notion that captures the knowledge of the adversary by pairs of symbolic traces, called bi-traces. A bi-trace is consistent when the messages in the two symbolic traces are indistinguishable by the adversary. Informally, two protocols are open bisimilar when each action in one protocol can be simulated in the other using a consistent bi-trace.

Diff-equivalence (d) is another strictly stronger notion that is defined for protocols that have the same structure and differ only by the messages they exchange. It means that, during execution, all communications and tests, including those that the adversary can make, either succeed for both protocols or fail for both protocols. This property implies that both protocols still have the same structure during execution.

Equational theories (S). What is the support for equational theories? At a high-level, extra support for certain axioms enables detecting a larger class of attacks (see, e.g., [40], [41]). We provide a coarse classification as follows: tools that support a fixed set of equational theories or no equational theories at all (○); tools that support user-defined equational theories, but without associative-commutative (AC) axioms (◐); tools that support user-defined equational theories with AC axioms (●). Supporting associative and commutative properties enables detecting a much larger class of attacks, since they allow the most detailed modeling of, e.g., xor operations, abelian groups, and Diffie-Hellman constructions. One caveat is that the finer details between these coarse classifications often make them incomparable, and even where they overlap, they are not all equally effective for analyzing concrete protocols.

Global mutable state (S). Does the tool support verification of protocols with global mutable state? Many real-world protocols involve shared databases (e.g., key servers) or shared memory, so reasoning support for analyzing complex, stateful attacks scenarios extends the reach of such tools [28].

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 analyze 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. Exceptions include scyther-proof [26], which generates proof scripts that can be machine-checked in the Isabelle theorem prover [42], and SPEC [36], which can produce explicit evidence of security claims that can be checked for correctness.

Specification language (U). How are protocols specified? The categorizations are domain-specific security protocol languages (▷), process calculus (⋆), multiset rewriting (∗), and general programming language (◇). General programming languages are arguably the most familiar to non-experts, while security protocol languages (i.e., notations for describing message flows between parties) are commonplace in cryptography. Process calculi and multiset rewriting may be familiar to formal methods practitioners. Process calculi are formal languages for describing concurrent processes and their interactions (e.g., [43]–[45]). Multiset rewriting is a more general and lower-level formalism that allows for various encodings of processes, but has no built-in notion of a process. It provides a natural formalism for complex state machines.

C. Symbolic Security: Discussion

Achievements: Symbolic proofs for real-world case studies. Of the considered symbolic tools, ProVerif and Tamarin stand out as having been used to analyze large, real-world protocols. They offer unprecedented combinations of scalability and expressivity, which enables them to deal with complex systems and properties. Moreover, they provide extensive documentation, a library of case studies, and practical usability features (e.g., packaging, a graphical user interface for Tamarin, attack reconstruction in HTML for Proverif).

Next, we provide a rough sense of their scalability on real-world case studies; more precise numbers can be found in the respective papers. It is important to keep in mind that comparisons between tools are difficult (even on similar case studies), so these numbers should be taken with a grain of salt.

ProVerif has been used to analyze TLS 1.0 [46] (seconds to several hours depending on the security property) and 1.3 [3] (around one hour), Signal [47] (a few minutes to more than a day depending on the security property), and Noise protocols [48] (seconds to days depending on the protocol). In general, more Diffie-Hellman key agreements (e.g., in Signal and Noise) increase analysis times.

Tool		RF	Auto	Comp	CS	Link	TCB
AutoG&P$^{\diamond}$	[55]	◐	●	○	◐	○	self, SMT
CertiCrypt$^{\triangleright\diamond}$	[56]	◑	○	○	●	●	Coq
CryptHOL$^{\diamond}$	[57]	◑	○	●	◐	○	Isabelle
CryptoVerif$^{\star\diamond}$	[58]	◐	●	○	●	●	self
EasyCrypt$^{\triangleright\diamond}$	[59]	◑	○	●	◐	●	self, SMT
F7$^{\diamond}$	[17]	◐	○	●	○	●	self, SMT
F*$^{\diamond}$	[60]	◐	○	●	○	●	self, SMT
FCF$^{\diamond}$	[61]	◑	○	●	◐	●	Coq
ZooCrypt$^{\diamond}$	[62]	◐	●	○	●	○	self, SMT

Reasoning Focus (RF)	Concrete security (CS)	Specification language
◐ – automation focus	● – security + efficiency	⋆ – process calculus
◑ – expressiveness focus	◐ – security only	▷ – imperative
	○ – no support	◇ – functional

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

Tamarin has been used to analyze the 5G authentication key exchange protocol [49] (around five hours), TLS 1.3 [2], [4] (around one week, requiring 100GB RAM), the DNP3 SAv5 power grid protocol [50] (several minutes), and Noise protocols [51] (seconds to hours depending on the protocol).

Challenge: Verifying equivalence properties. Many security properties can be modeled accurately by equivalence properties, but they are inherently more difficult to verify than trace properties. This is because they involve relations between traces instead of single traces. As such, tool support for reasoning about equivalence properties is thus substantially less mature. For full automation, 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.

For the bounded setting, recent developments include support for more equational theories (AKISS [32], DEEPSEC [34]), for protocols with else branches (APTE [33], AKISS, DEEPSEC) and for protocols whose actions are not entirely determined by their inputs (APTE, DEEPSEC). There have also been performance improvements based on partial order reduction (APTE, AKISS, DEEPSEC) or graph planning (SAT-Equiv). For the unbounded setting, diff-equivalence, first introduced in ProVerif [52] and later adopted by Maude-NPA [53] and Tamarin [54], remains the only fully automated approach for proving equivalences. Because trace equivalence is the most natural for formalizing privacy properties, verifying more 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.

Reasoning focus (U). Is the tool's reasoning focus on automation (◐) or on expressivity (◑)? Automation focus means being able to produce automatically or with light interaction a security proof (at the cost of some expressiveness). Dually, expressivity focus means being able to express arbitrary arguments (at the cost of some automation).

Automated proof-finding (U). Can the tool automatically find security proofs? A subset of the automation-focused tools can automatically (non-interactively) find security proofs in restricted settings (e.g., proofs of pairing-based schemes for AutoG&P, proofs of key exchange protocols using a catalog of built-in game transformations for CryptoVerif, proofs of padding-based public key encryption schemes for ZooCrypt).

Composition (U). Does the tool support compositional reasoning? Support for decomposing security arguments of cryptographic systems into security arguments for their core components is essential for scalable analysis.

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

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)? An established general-purpose theorem prover such as Coq [63] or Isabelle [64] is usually held as the minimum TCB for proof checking. Most tools, however, rely on an implementation of the tool's logics in a general purpose language that must be trusted (self). Automation often relies on SMT solvers [65], such as Z3 [66].

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 been used to develop machine-checked security proofs for a range of real-world cryptographic mechanisms. CryptoVerif has been used for a number of protocols, including TLS 1.3 [3], Signal [47], and WireGuard [67]. EasyCrypt has been used for the Amazon Web Service (AWS) key management system [68] and the SHA-3 standard [69]. F7 was used to build miTLS, a reference implementation of TLS 1.2 with verified computational security at the code-level [70], [71]. 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 π-calculus [45], 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 cryptographic core. Computational proofs in F* rely on transforming a detailed protocol description into a final (ideal) program by relying on ideal functionalities for cryptographic primitives. Formal validation of this transformation is carried out manually, with some help from the F* verification infrastructure. Formal verification of the final program is done 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 [72], [73] as well as in verifying their concrete design and implementation [1], [70].

Takeaway: EasyCrypt is the closest to pen-and-paper cryptographic proofs. EasyCrypt supports a general-purpose relational program logic (i.e., a formalism for specifying and verifying properties about two programs or two runs of the same program) that captures many of the common game hopping techniques. This is complemented by libraries that support other common techniques, e.g., the PRF/PRP switching lemma, hybrid arguments, and lazy sampling [8]. In addition, EasyCrypt features a union bound logic for upper bounding the probability of some event E in an experiment (game) G (e.g., bounding the probability of collisions in experiments that involve hash functions). Overall, EasyCrypt proofs closely follow the structure of pen-and-paper arguments. A consequence is that EasyCrypt is amenable to proving the 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 [74]. Most game-based security definitions do not provide out-of-the-box composition guarantees, so simulation-based definitions are the preferred choice for analyzing large cryptographic systems, with universal composability (UC) being the gold-standard—UC definitions guarantee secure composition in arbitrary contexts [9]. Work on developing machine-checked UC proofs is relatively nascent [75]–[77], but is an important and natural next step for computational tools.

F. Further Reading

Another class of tools leverages the benefits of automated verification to support automated synthesis of secure cryptographic designs, mainly in the computational world [62], [78]–[81]. Cryptographic compilers provide high-level abstractions (e.g., a domain-specific language) for describing cryptographic tasks, which are then compiled into custom protocol implementations. These have been proposed for verifiable computation [82]–[85], zero-knowledge [86]–[89], and secure multiparty computation [90] protocols, which are parameterized by a proof-goal or a functionality to compute. Some are supported by proofs that guarantee the output protocols are correct and/or secure for every input specification [91]–[94]. We recommend readers to also consult other related surveys. Blanchet [95] surveys design-level security until 2012 (with a focus on ProVerif). Cortier et al. [96] 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, 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 overall 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 ultimately produces highly tuned, platform-specific assembly code [103]. Many more examples of high-speed crypto code written at assembly and pre-assembly levels can be found in SUPERCOP [107], a benchmarking framework for cryptography implementations.

More broadly, efficiency considerations typically rule out exclusively 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 [108]. 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 [109]; an arithmetic bug in OpenSSL led to a full key recovery attack [110]). Current solutions for preventing more mistakes 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 `libcrypto` library reported 24 vulnerabilities between January 1, 2016 and May 1, 2019 [7].

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, without compromising efficiency.

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 may be carried out on a verification-friendly representation of the source code, rather than on the source code itself. 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 implementation can be proved functionally correct in two different ways: equivalence to a reference implementation, or satisfying a functional specification, typically expressed as preconditions (what the program requires on inputs) and postconditions (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 their correctness proofs often rest on non-trivial mathematics. Mechanizing them thus requires 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.

Typically, functional correctness proofs are carried out at source level. A long-standing challenge is how to carry guarantees to machine code. This can be addressed using verified compilers, which are supported by formal correctness proofs. CompCert [111] is a prime example of moderately optimizing verified compiler for a large fragment of C. However, the

Tool		Memory safety	Automation	Parametric verification	Input language	Target(s)	TCB
Cryptol + SAW	[97]	●	◐	○	C, Java	C, Java	SAT, SMT
CryptoLine	[98]	○	●	○	CryptoLine	C	Boolector, MathSAT, Singular
Dafny	[99]	●	◐	○	Dafny	C#, Java, JavaScript, Go	Boogie, Z3
F*	[60]	●	◐	○	F*	OCaml, F#, C, Asm, Wasm	Z3, typechecker
Fiat Crypto	[6]	●	○	●	Gallina	C	Coq, C compiler
Frama-C	[100]	●	◐	○	C	C	Coq, Alt-Ergo, Why3
gfverif	[101]	○	●	○	C	C	g++, Sage
Jasmin	[102]	●	◐	○	Jasmin	Asm	Coq, Dafny, Z3
Vale	[103], [104]	●	◐	●	Vale	Asm	Dafny or F*, Z3
VST	[105]	●	○	○	Gallina	C	Coq
Why3	[106]	○	◐	○	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.

trade-off is that verified compilers typically come with fewer optimizations than mainstream compilers and target fewer platforms.

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 (○).

Parametric verification (U). Can the tool verify parameterized implementations? This enables writing and verifying generic code that can be used to produce different implementations depending on the supplied parameters. For example, Fiat Crypto [6] can generate verified elliptic curves implementations parameterized by a prime modulus, limb representation of field elements, and hardware platform; Vale [103], [104] implementations are parameterized by the operating system, assembler, and hardware platform.

Input language (U). What is the input language? Many toolchains use custom verification-oriented languages. Dafny is a high-level imperative language, whereas F*, Gallina (used in Coq), and WhyML (used in Why3) are functional languages. CryptoLine, Jasmin, and Vale are assembly-like languages; Jasmin and Vale provide high-level control-flow structures such as procedures, conditionals, and loops. Other tools take code written in existing languages (e.g., C, Java).

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 [111]) 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 building-

Implementation		FC	CT	Tool(s)	Target	% faster
evercrypt	[7]	●	●	F*, Vale	64-bit C, Intel ADX asm	25.92
precomp	[112]	○	○	–	Intel ADX asm	25.77
sandy2x	[113]	○	○	–	Intel AVX asm	11.15
hacl	[7]	●	●	F*	64-bit C	8.69
jasmin	[102]	○	●	Jasmin	Intel x86_64 asm	7.88
amd64	[114]	◐	○	Coq, SMT	Intel x86_64 asm	6.11
fiat	[6]	●	○	Fiat Crypto	64-bit C	5.39
donna64	[115]	○	○	–	64-bit C	0.00

Functional correctness (FC), Constant-time (CT)
● – verified ◐ – partially verified ○ – not verified

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

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 being deployed at scale. Verified primitives in the HACL* [5] library are used in Mozilla Firefox's NSS security engine, and verified elliptic curve implementations in the Fiat Cryptography library [6] are used in Google's BoringSSL library.

There are several common insights to these successes. 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 use 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. However, various projects in the cryptography domain have challenged this position. We are seeing verified implementations that meet the performance of the fastest unverified implementations. We conclude that there is currently no conceptual or technological barrier that prevents verifying the fastest implementations available, although more effort is expected.

As a small case study, we look at Curve25519 [116], 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 compare their performance, 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 [115], one of the fastest known (unverified) C implementations. All measurements are collected on a 1.8 GHz Intel i7-8565U CPU 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 optimization flag -O3. To summarize, several verified C implementations (hacl and fiat) beat donna64; the fastest verified assembly implementation (evercrypt) meets the fastest unverified assembly implementation (precomp).

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 efficient, but verification becomes much easier (e.g., memory safety can be obtained for free). These aspects are discussed further in the Vale and Jasmin papers [102], [103], [117].

Challenge: Automating equivalence proofs. Significant progress could be made if functional correctness proofs could be solved by providing a sequence of simple transformations that connect specifications to targets and relying on an automatic tool to check these simple transformations. Promising recent work in this direction [118] demonstrates the feasibility of the approach. However, the current approaches are not automatic: neither in finding the transformations nor in proving them. 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 often involves tricky mathematical reasoning that SMT-based tools can struggle with. Examples range from proving the correctness of the Montgomery representations [119] 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 [120]) 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 [121]–[127], distributed systems [128]–[130], and even entire software stacks [131]. 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 software protections against digital side-channel attacks. Hardware protections are beyond the scope of this paper and are left as further reading. By digital side-channel attacks, we mean those that can be launched 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 are devastating—key-recovery attacks have been demonstrated on real implementations, e.g., on RSA [142] and AES [143].

How can implementation-level security fail? The prevailing technique for protecting against digital side-channel attacks is to follow *constant-time* coding guidelines [144]. 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 implementations must

Tool		Target	Method	Synthesis	Sound	Complete	Public inputs	Public outputs	Control flow	Memory access	Variable-time op.
ABPV13	[132]	C	DV	○	●	●	●	○	●	●	○
CacheAudit	[133]	Binary	Q	○	●	○	○	○	●	●	○
ct-verif	[134]	LLVM	DV	○	●	●	●	●	●	●	●
CT-Wasm	[135]	Wasm	TC	○	●	○	●	○	●	●	●
FaCT	[136]	LLVM	TC	●	●	○	●	○	●	●	●
FlowTracker	[137]	LLVM	DF	○	●	○	●	○	●	●	○
Jasmin	[102]	asm	DV	○	●	●	●	●	●	●	○
KMO12	[138]	Binary	Q	○	●	○	○	○	○	●	○
Low*	[139]	C	TC	○	●	○	●	○	●	●	●
SC Eliminator	[140]	LLVM	DF	●	●	○	●	○	●	●	○
Vale	[103]	asm	DF	○	●	○	●	●	●	●	●
VirtualCert	[141]	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.

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 implementors 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 instruction set architecture (ISA) optimizations, can remove source-level countermeasures. 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 [145] led to the Lucky 13 timing vulnerability in OpenSSL [146]; in turn, the Lucky 13 patch led to yet another timing vulnerability [147]!

To prevent compiler optimizations from interfering with constant-time recipes applied at the source-code level, implementors 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 [148], [149], 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. These approaches necessarily abstract the leakage interface available to real-world attackers, but being precisely defined, they help clarify the gap 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 program 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*, an information flow property requiring that variations in secret data cause no differences in observable outputs [150].

The simplest leakage model is the program counter policy, where the program control-flow is leaked during execution [151]. The most common leakage model, namely the constant-time policy, 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 launched by co-located processes. A more precise leakage model called the size-respecting policy also assumes that operand sizes are leaked for specific variable-time operations. For more information on leakage models, see the paper by Barthe et al. [150, Section IV.D].

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

Table V 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 construct a rich model of a hardware feature, e.g, the cache, and derive an upper-bound on the leaked information.
- Deductive verification techniques 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 on 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 [136] and SC Eliminator [140]) 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. Still, it bears mentioning that some unsound tools are used in practice. One example is ctgrind [152], an extension of Valgrind that takes in a binary with taint annotations and checks for constant-address security via dynamic analysis. It supports public inputs but not public outputs, and is neither sound nor complete.

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 values associated with conditional branching (e.g., if, switch, while, for statements) during program execution.

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

Variable-time operation leakage (S). Does the tool consider variable-time operation leakage? The leakage model includes inputs to variable-time operations (e.g., floating point operations [153]–[155], division and modulus operations on some architectures) 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 [103], high-speed implementations of SHA-3 and TLS 1.3 ciphersuites have been verified using Jasmin [102], and various off-the-shelf libraries have been analyzed with FlowTracker [137].

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 scrutiny, 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 [156], 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. [150] laid the theoretical foundations of constant-time preserving compilation. These ideas were subsequently realized in the verified CompCert C compiler [157].

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 [148] and Meltdown [149]. 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 ISA describes (usually informally) what one needs to know to write a functionally correct program [158], [159]. However, current ISAs are an insufficient specification of the hardware-software contract when it comes to writing secure programs [160]. 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 [160]. 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 several threads of relevant work, e.g., on verifying side-channel resistance in hardware [161]–[165], and on verifying masked implementations aimed at protecting against differential power analysis attacks [166]–[171].

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 are 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. [69] formally verify an efficient implementation of the sponge construction from the SHA-3 standard. It connects proofs of random oracle (RO) 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. [174] 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 correctly 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. Many

Implementation(s)	Target(s)	Tool(s) used	Computational security	Functional correctness	Efficiency	Side-channel resistance
RSA-OEAP [172]	C	EasyCrypt, Frama-C, CompCert	●	●	○	●
Curve25519 scalar mult. loop [114]	asm	Coq, SMT	–	●	●	○
SHA-1, SHA-2, HMAC, RSA [131]	asm	Dafny, BoogieX86	–	●	●	○
HMAC-SHA-2 [173]	C	FCF, VST, CompCert	●	●	○	○
MEE-CBC [174]	C	EasyCrypt, Frama-C, CompCert	●	●	○	●
Salsa20, AES, ZUC, FFS, ECDSA, SHA-3 [175]	Java, C	Cryptol, SAW	○	◐	◐	○
Curve25519 [176]	OCaml	F*, Sage	–	◐	○	◐
Salsa20, Curve25519, Ed25519 [102]	asm	Jasmin	○	○	●	●
SHA-2, Poly1305, AES-CBC [103]	asm	Vale	○	●	○	●
HMAC-DRBG [177]	C	FCF, VST, CompCert	●	●	○	○
HACL*[1] [5]	C	F*	◐	◐	◐	◐
HACL*[1] [5]	C	F*, CompCert	◐	●	○	●
HMAC-DRBG [178]	C	Cryptol, SAW	○	◐	◐	○
SHA-3 [69]	asm	EasyCrypt, Jasmin	●	●	●	●
ChaCha20, Poly1305 [117]	asm	EasyCrypt, Jasmin	○	●	●	●
BGW multi-party computation protocol [179]	OCaml	EasyCrypt, Why3	●	◐	○	○
Curve25519, P-256 [6]	C	Fiat Crypto	–	◐	◐	○
Poly1305, AES-GCM [104]	asm	F*, Vale	○	●	●	●
Bignum code[4] [98]	C	CryptoLine	–	●	◐	○
WHACL*[1], LibSignal* [180]	Wasm	F*	◐	●	◐	●
EverCrypt[2] [7]	C	F*	○	◐	◐	◐
EverCrypt[3] [7]	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.

implementations target *either* C *or* assembly. This involves trade-offs between the portability and lighter verification-effort of C code, and the efficiency that can be gained via hand-tuned assembly. EverCrypt [7] 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 [7], [104].

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 cryptographic cores. However, once the academic community started considering more detailed aspects of the protocol, many new attacks were discovered, e.g., [181], [182].

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 [183] 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. The work surrounding TLS has shown 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. [70] discovered new alert fragmentation and fingerprinting attacks and led to the discovery of the Triple Handshake attacks [72]. 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 analysis of TLS 1.3 using ProVerif by Bhargavan et al. [3] uncovered a new attack on 0-RTT client authentication that was fixed in draft 13. The symbolic analysis of 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 and reliably 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 claims. On 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. [173] 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 are useful for tracking progress over time when used appropriately. The HACL* [5] study 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 modeling a problem in the exact same way is non-trivial.

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 allow the artifacts (e.g., machine-checked proofs) to become stale, or else muster significant human effort 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 functionality, or in the worst case, to swiftly patch new vulnerabilities beyond what was verifiably accounted for. To this end, we hope to see more interplay between proof engineering research [184], [185] and computer-aided cryptography research in the coming years.

C. Recommendations to Standardization Bodies

Given its benefits in the TLS 1.3 standardization effort, we believe computer-aided cryptography should play an important role in standardization processes [186]. Traditionally, cryptographic standards are written in a combination of prose, formulas, and pseudocode, and can change drastically between drafts. On top of getting the cryptography right in the first place, standards must also focus on clarity, ease of implementation, and interoperability. Unsurprisingly, standardization processes can be long and arduous. And even when they are successful, the substantial gap between standards and implementations leaves plenty of room 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 believe the latter is a lesser evil, as it does not create a false sense of security. For example, Hales [187] discusses ill-intentioned security arguments in the context of the Dual EC pseudo-random generator [188]. Another example is the recent discovery of attacks against the AES-OCB2 ISO standard, which was previously believed to be secure [189].

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 [190] as an excellent opportunity to put our recommendations into practice, and we encourage the computer-aided cryptography community to engage in the process.

Acknowledgments

We thank the anonymous reviewers for their useful suggestions; Jason Gross, Boris Köpf, Stever Kremer, Peter Schwabe, and Alwen Tiu for feedback on earlier drafts of the paper; and Tiago Oliveira for help setting up Jasmin and benchmarks.

Work by Manuel Barbosa was supported by National Funds through the Portuguese Foundation for Science and Technology (FCT) under project PTDC/CCI-INF/31698/2017. Work by Gilles Barthe was supported by the Office of Naval Research (ONR) under project N00014-15-1-2750. Work by Karthik Bhargavan was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 683032 - CIRCUS). Work by Bruno Blanchet was supported by the French National Research Agency (ANR) under project TECAP (decision no. ANR-17-CE39-0004-03). Work by Kevin Liao was supported by the National Science Foundation (NSF) through a Graduate Research Fellowship. Work by Bryan Parno was supported by a gift from Bosch, a fellowship from the Alfred P. Sloan Foundation, the NSF under Grant No. 1801369, and the Department of the Navy, Office of Naval Research under Grant No. N00014-18-1-2892.

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 *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 *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 *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 *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 *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 *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 1202–1219.

[7] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin, "EverCrypt: A fast, verified, cross-platform cryptographic provider," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.

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

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

[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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. LNCS, 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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. LNCS, 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] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *IACR Cryptology ePrint Archive*, vol. 2004, p. 332, 2004. [Online]. Available: http://eprint.iacr.org/2004/332

[15] 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.

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

[17] 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.

[18] 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.

[19] 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. LNCS, vol. 5705. Springer, 2007, pp. 1–50.

[20] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

[21] 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.

[22] V. Cheval, V. Cortier, and M. Turuani, "A little more conversation, a little less action, a lot more satisfaction: Global states in proverif," in *IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2018, pp. 344–358.

[23] D. L. Li and A. Tiu, "Combining proverif and automated theorem provers for security protocol verification," in *International Conference on Automated Deduction (CADE)*, ser. LNCS, vol. 11716. Springer, 2019, pp. 354–365.

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

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

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

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

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

[29] M. Turuani, "The cl-atse protocol analyser," in *International Conference on Term Rewriting and Applications (RTA)*, ser. LNCS, vol. 4098. Springer, 2006, pp. 277–286.

[30] 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.

[31] A. Armando and L. Compagna, "SATMC: A sat-based model checker for security protocols," in *European Conference on Logics in Artificial*

Intelligence (JELIA), ser. LNCS, vol. 3229. Springer, 2004, pp. 730–733.

[32] 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.

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

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

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

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

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

[38] 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.

[39] 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.

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

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

[42] L. C. Paulson, *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, ser. LNCS. Springer, 1994, vol. 828.

[43] R. Milner, *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[44] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The spi calculus," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 1997, pp. 36–47.

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

[46] 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.

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

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

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

[50] 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.

[51] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. Basin, "A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols," in *Proc. of USENIX Security*, 2020.

[52] 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.

[53] 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. LNCS, vol. 8743. Berlin, Heidelberg: Springer, Sep. 2014, pp. 162–177.

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

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

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

[57] 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.

[58] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, Oct.–Dec. 2008.

[59] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *International Cryptology Conference (CRYPTO)*, ser. LNCS, vol. 6841. Springer, 2011, pp. 71–90.

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

[61] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *International Conference on Principles of Security and Trust (POST)*, ser. LNCS, vol. 9036. Springer, 2015, pp. 53–72.

[62] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 1247–1260.

[63] "The coq proof assistant." [Online]. Available: https://coq.inria.fr/

[64] "Isabelle." [Online]. Available: https://isabelle.in.tum.de/

[65] C. W. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[66] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

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

[68] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 63–78.

[69] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 1607–1622.

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

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

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

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

[74] C. E. Landwehr, D. Boneh, J. C. Mitchell, S. M. Bellovin, S. Landau,

and M. E. Lesk, “Privacy and cybersecurity: The next 100 years,” *Proc. of the IEEE*, vol. 100, no. Centennial-Issue, pp. 1659–1673, 2012.

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

[76] R. Canetti, A. Stoughton, and M. Varia, “EasyUC: Using EasyCrypt to mechanize proofs of universally composable security,” in *IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 167–183.

[77] A. Lochbihler, S. R. Sefidgar, D. A. Basin, and U. Maurer, “Formalizing constructive cryptography using CryptHOL,” in *IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 152–166.

[78] 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.

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

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

[81] 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.

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

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

[84] S. T. V. 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 *USENIX Security Symposium (USENIX)*. USENIX Association, 2012, pp. 253–268.

[85] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, “SNARKs for C: verifying program executions succinctly and in zero knowledge,” in *International Cryptology Conference (CRYPTO)*, ser. LNCS, vol. 8043. Springer, 2013, pp. 90–108.

[86] 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 *European Symposium on Research in Computer Security (ESORICS)*, 2010, pp. 151–167.

[87] M. Fredrikson and B. Livshits, “Zø: An optimizing distributing zero-knowledge compiler,” in *USENIX Security Symposium (USENIX)*, 2014, pp. 909–924.

[88] 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 *USENIX Security Symposium (USENIX)*. USENIX Association, 2010, pp. 193–206.

[89] M. Backes, M. Maffei, and K. Pecina, “Automated synthesis of secure distributed applications,” in *Symposium on Network and Distributed System Security (NDSS)*. The Internet Society, 2012.

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

[91] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 488–500.

[92] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1989–2006.

[93] C. Fournet, C. Keller, and V. Laporte, “A certified compiler for verifiable computing,” in *IEEE Computer Security Foundations Symposium (CSF)*, 2016, pp. 268–280.

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

[95] B. Blanchet, “Security protocol verification: Symbolic and computational models,” in *International Conference on Principles of Security and Trust (POST)*, ser. LNCS, vol. 7215. Springer, 2012, pp. 3–29.

[96] 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.

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

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

[99] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, ser. LNCS, vol. 6355. Springer, 2010, pp. 348–370.

[100] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-c - A software analysis perspective,” in *International Conference on Software Engineering and Formal Methods (SEFM)*, ser. LNCS, vol. 7504. Springer, 2012, pp. 233–247.

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

[102] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1807–1823.

[103] 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 *USENIX Security Symposium (USENIX)*. USENIX Association, 2017, pp. 917–934.

[104] 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.

[105] A. W. Appel, “Verified software toolchain - (invited talk),” in *European Symposium on Programming (ESOP)*, ser. LNCS, vol. 6602. Springer, 2011, pp. 1–17.

[106] J. Filliâtre and A. Paskevich, “Why3 - where programs meet provers,” in *European Symposium on Programming (ESOP)*, ser. LNCS, vol. 7792. Springer, 2013, pp. 125–128.

[107] D. J. Bernstein and T. Lange, “ebacs: Ecrypt benchmarking of cryptographic systems,” 2009. [Online]. Available: https://bench.cr.yp.to

[108] 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 *Internet Measurement Conference (IMC)*. ACM, 2014, pp. 475–488.

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

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

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

[112] 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 *International Conference on Selected Areas in Cryptography (SAC)*, ser. LNCS, vol. 10719. Springer, 2017, pp. 172–191.

[113] T. Chou, “Sandy2x: New curve25519 speed records,” in *International Conference on Selected Areas in Cryptography (SAC)*, ser. LNCS, vol. 9566. Springer, 2015, pp. 145–160.

[114] Y. Chen, C. Hsu, H. Lin, P. Schwabe, M. Tsai, B. Wang, B. Yang, and S. Yang, “Verifying curve25519 software,” in *ACM Conference*

on Computer and Communications Security (CCS). ACM, 2014, pp. 299–309.

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

[116] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *IACR International Conference on Practice and Theory of Public-Key Cryptography (PKC)*, ser. LNCS, vol. 3958. Springer, 2006, pp. 207–228.

[117] 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.

[118] 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.

[119] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

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

[121] 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.

[122] 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 *Symposium on Principles of Programming Languages (POPL)*. ACM, 2015, pp. 595–608.

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

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

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

[126] 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 *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 18–37.

[127] 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 *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2013, pp. 430–444.

[128] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2015, pp. 357–368.

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

[130] 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 *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 1–17.

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

[132] 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.

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

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

[135] 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.

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

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

[138] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *International Conference on Computer-Aided Verification (CAV)*, ser. LNCS, vol. 7358. Springer, 2012, pp. 564–580.

[139] 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.

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

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

[142] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *USENIX Security Symposium (USENIX)*. USENIX Association, 2003.

[143] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

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

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

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

[147] 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

[148] 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 *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 1–19.

[149] 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 *USENIX Security Symposium (USENIX)*. USENIX Association, 2018, pp. 973–990.

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

[151] 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 *International Conference on Information Security and Cryptology (ICISC)*, ser. LNCS, vol. 3935. Springer, 2005, pp. 156–168.

[152] A. Langley, "ctgrind," 2010. [Online]. Available: https://github.com/agl/ctgrind/

[153] M. Andrysco, A. Nötzli, F. Brown, R. Jhala, and D. Stefan, "Towards verified, constant-time floating point operations," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 1369–1382.

[154] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2015, pp. 623–639.

[155] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security Symposium (USENIX)*. USENIX Association, 2017, pp. 69–81.

[156] 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 *International Conference on Cryptology and Network Security (CANS)*, ser. LNCS, vol. 10052, 2016, pp. 573–582.

[157] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 7:1–7:30, 2020.

[158] 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.

[159] 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.

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

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

[162] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009, pp. 109–120.

[163] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014, pp. 97–112.

[164] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 109–120.

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

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

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

[168] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 7428. Springer, 2012, pp. 58–75.

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

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

[171] 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 *European Symposium on Research in Computer Security (ESORICS)*, ser. LNCS, vol. 11735. Springer, 2019, pp. 300–318.

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

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

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

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

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

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

[178] 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 *International Conference on Computer-Aided Verification (CAV)*, ser. LNCS, vol. 10982. Springer, 2018, pp. 430–446.

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

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

[181] C. Meyer and J. Schwenk, "Sok: Lessons learned from SSL/TLS attacks," in *Proc. of the International Workshop on Information Security Applications (WISA)*, ser. LNCS, vol. 8267. Springer, 2013, pp. 189–209.

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

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

[184] 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.

[185] 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.

[186] K. Bhargavan, F. Kiefer, and P. Strub, "hacspec: Towards verifiable crypto standards," in *International Conference on Security Standardisation Research (SSR)*, ser. LNCS, vol. 11322. Springer, 2018, pp. 1–20.

[187] T. C. Hales, "The nsa back door to nist," *Notices of the AMS*, vol. 61, no. 2, pp. 190–192, 2014.

[188] 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 *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 468–479.

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

[190] 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.