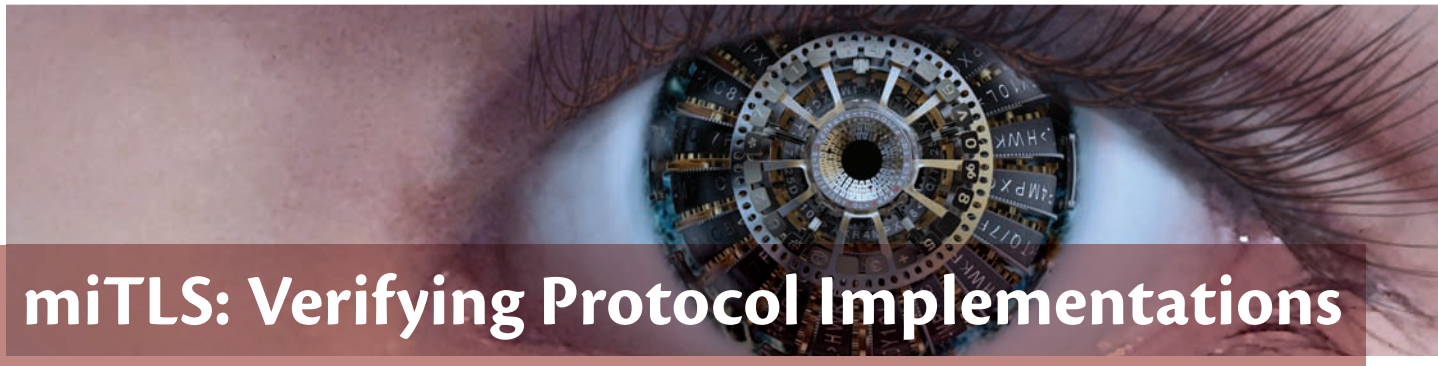

miTLS: Verifying Protocol Implementations against Real-World Attacks

Karthikeyan Bhargavan | INRIA
Cédric Fournet and Markulf Kohlweiss | Microsoft

TLS is the default protocol for encrypting communications between web clients and servers. Despite analysis to establish strong formal guarantees for various configurations, TLS deployments are often vulnerable to real-world attacks. The miTLS project examines this gap between the theory and practice of TLS.

The TLS Internet standard, previously known as SSL, is the default protocol for encrypting communications between clients and servers on the web. Hence, TLS routinely protects our sensitive emails, health records, and payment information against network-based eavesdropping and tampering. For the past 20 years, TLS security has been analyzed in a variety of cryptographic and programming models to establish strong formal guarantees for various configurations of the protocol. However, TLS deployments are still often found to be vulnerable to attacks and rely on security experts to fix the protocol implementations.

TLS Origins

Both the Internet and cryptography took roots in military technology. One of the first uses of computers, pioneered by Alan Turing, was to decrypt German wartime communications, and the Internet's precursor, the Arpanet, was designed for resilience in case of nuclear war. It was the invention of public-key cryptography by Whitfield Diffie and Martin Hellman that created the impetus for open academic research into cryptography and eventually led to the ubiquitous use of encryption on the Internet.

The SSL protocol, one of the first real-world deployments of public-key cryptography, was originally developed by Netscape, an early Internet browser vendor, to provide secure channels for electronic commerce. One of its main designers was Taher Elgamal, a student of Hellman. As SSL took over the web, it was renamed TLS and documented as an open standard by the Internet Engineering Task Force (IETF). Over time, it has undergone major changes; its implementations currently feature five versions—SSL2, SSL3, TLS 1.0, TLS 1.1, and TLS 1.2—while the IETF is actively discussing the protocol's next version.

TLS implements a network socket API on top of a reliable but insecure network. It consists of two main protocols:

- a handshake that establishes sessions between clients and servers, relying on public-key cryptography to compute shared session keys, and
- a record layer that uses those keys to encrypt and authenticate their communications.

 Copublished by the IEEE Computer and Reliability Societies

SSL2 initially supported a single handshake scheme, based on RSA encryption and a few record encryption algorithms, such as Data Encryption Standard (DES) and Rivest Cipher 2 (RC2; Ronald L. Rivest is also the R in RSA.) SSL3 added Diffie-Hellman schemes to the handshake as well as further encryption algorithms, such as RC4 and 3DES. Over time, many of these cryptographic constructions came under attack and were supplemented with stronger alternatives.

Because the client and the server might support different sets of cryptographic algorithms, the handshake lets them negotiate a combination of algorithms, called a *ciphersuite*. Hence, any TLS client and server can interoperate as long as they have at least one ciphersuite in common. The number of ciphersuites supported by TLS implementations has grown steadily over time. For example, the popular OpenSSL library now supports more than 100 ciphersuites.

Not all ciphersuites are equally strong. Like most commercial cryptographic software during the Cold War, SSL was subject to US export regulations that classified cryptography as a weapon. To comply with these regulations, all protocol versions up to TLS 1.0 included deliberately weakened encryption algorithms for use in US software, such as web browsers, exported to foreign countries. Cryptographers and security practitioners started a rebellion, dubbed the Crypto Wars, against this weakening of their work, and eventually prevailed, but SSL and TLS implementations were still forced to support export-grade ciphersuites for interoperability.

Many of the challenges in designing and deploying TLS securely were already apparent in the early days of the protocol. In particular, Daniel Bleichenbacher demonstrated a side-channel attack against the way RSA encryption was used in the SSL handshake, and Serge Vaudenay discovered another side-channel attack on the way application data was encrypted in the record protocol.[1] Later versions of TLS continued to support those weak constructions but mandated that implementations employ adequate countermeasures, triggering a series of increasingly sophisticated attacks and defenses.

Besides cryptographic weaknesses, the SSL handshake protocol itself was shown to be vulnerable to logical flaws. The negotiation between strong and weak encryption had a protocol-level flaw in SSL2: a ciphersuite rollback (or downgrade) identified by Martin Abadi,[2] enabling a network attacker to force a client and a server to use a weak export ciphersuite even though they both preferred a stronger ciphersuite. This flaw was fixed in SSL3, but a subsequent analysis revealed a more advanced downgrade attack,[3] enabling a network attacker to first force SSL3 clients and servers to use SSL2, and then exploit its known weaknesses. This was fixed by modifying the use of RSA encryption, which in turn enabled an improved Bleichenbacher-style side-channel attack.

Hence, by the early 2000s, TLS was already caught in a cycle of attacks and fixes that continues today. Formal foundations to validate the protocol design and prevent any such attacks became very attractive, and researchers from both the cryptographic and formal methods communities started applying various verification techniques to communications protocols.

Provable Security and Symbolic Verification

Since the 1980s, cryptographers had been working on turning cryptography from an art into a science. The resulting theory is nowadays referred to as *provable security*. Conceived in MIT's Theory of Computation group, it attempts to reduce the difficulty of breaking cryptographic protocols to problems in complexity theory and mathematics. This approach resulted in groundbreaking work such as that of the Turing award winners Shafi Goldwasser and Silvio Micali on probabilistic encryption and zero-knowledge proofs.

From the cryptographer's point of view, the TLS protocol is a combination of standard cryptographic constructions. Using compositional provable security techniques, we should be able to prove the security of each construction, and then put these proofs together to obtain a security theorem for TLS. In reality, composing proofs of various ad hoc parts of the protocol turned out to be hard, but over the past decade, cryptographers have successfully analyzed the security of many popular TLS ciphersuites. Their theorems confirm that, under some well-defined implementation and mathematical assumptions, the cryptographic core of TLS isn't vulnerable to attack.

From the programmer's point of view, protocols like TLS can be viewed as distributed processes that communicate across public channels and use cryptographic primitives as black boxes to protect their messages. The key analysis question is then whether the protocol, seen as a program, has logical flaws in its use of communications and cryptography, even if one assumes that the cryptographic building blocks are perfectly secure. For example, we might ask whether TLS admits ciphersuite or version downgrade attacks in the presence of an active network adversary.

The verification of concurrent and distributed processes has been investigated in a long line of research on programming language semantics, pioneered by Robin Milner and Tony Hoare, who were also Turing award winners. Their rigorous mathematical study of the meaning of programs lets us formalize what it means for a program to keep a value secret, or for two programs to be equivalent. For simple cryptographic primitives,

modeled as abstract mathematical functions, a message can hide a secret if it doesn't visibly depend on it, and two processes might be equivalent if they exchange similar-looking messages.

Most cryptographic algorithms hide secrets only computationally, meaning that given sufficient computational resources, the secret can eventually be recovered. Their precise modeling requires complicated probabilistic definitions against restricted classes of adversaries. Instead, the semantics community proposed simpler symbolic approximations of cryptography to capture logical flaws and developed tools to prove security (or find attacks) in their models. These techniques were used in a series of automated analyses of TLS, showing that the protocol isn't vulnerable to logical attack, as long as the attacker can't break the cryptographic primitives.

Both the provable security and the symbolic verification of protocols were successful in their respective academic communities. As the former is more precise and the latter easier to automate, they are in principle complementary. However, the technical differences outlined earlier led to largely separate developments. As we will argue, this limited the impact either of them had on real-world TLS security.

Theory versus Practice

Despite these theoretical successes, recent TLS versions have still been found vulnerable to practical attacks that rely on a combination of implementation bugs, cryptographic weaknesses, and protocol flaws. These attacks make it evident that the provable security and verification communities' most advanced models still ignored many important implementation details. These include message formats, support for multiple protocol modes and algorithms for backward compatibility, error handling exploitable as side channels, and signaling between the protocol and the application. Because these details affect practical TLS security, their omission limits the scope of theoretical statements.

It's worth reflecting on the cultural reasons for this gap between theory and practice. In their 2011 article on provable security, Jean Paul Degabriele and his colleagues explain that a focus on principles can lead to simplistic or artificial models as well as the neglect of implementation details.[4] Interestingly, they notice a similar divide in the practical security community between specification writers and implementers. The former build flexibility into specifications to allow for the competing interests of parties contributing to the development process. For example, specifications often avoid defining an API and encourage implementations to accept a broad range of behaviors to support interoperability and backward compatibility. This flexibility can tempt cryptographers to interpret specifications in an overly abstract way that facilitates security analysis but misses real-world attacks that rely on implementation details.

Instead, we follow a model-attack-remodel cycle, informed by a dialog between practitioners and theoreticians. Concrete attack scenarios are invaluable for practice-oriented provable security: if they fall outside the security model, they encourage researchers to refine their model to better account for realistic threats. Conversely, model features that don't reflect any such scenario could point out simplifications.

Let us also mention a class of attacks often missed by practitioners and theoreticians alike. These attacks target the protocol design and evaluation process, sometimes directly, through the insertion of back doors or, more subtly, through influence on the culture in which designers operate. Juniper's VPN security hole is a recent example in this class. Besides awareness of the interests that some organizations might have in subverting Internet security, we believe that formal, open, practice-oriented protocol verification helps prevent such attacks.

A more technical challenge that prevents cryptographic analysis techniques on TLS deployments is that the protocol and its implementations have grown too complicated to be analyzed by humans. Unsurprisingly, the highly optimized C code of TLS implementations such as OpenSSL is amenable neither to cryptographic proofs nor to formal verification.

Cryptographic and symbolic TLS models alike couldn't keep up with implementations and didn't account for the protocol details specified in the standard. Consequently, proofs of these models were likely to miss practical attacks on the protocol. Of course, they also missed attacks that exploited basic implementation flaws, such as incorrect certificate validation (GotoFail) or buffer overflows (HeartBleed).

In summary, we argue that the coexistence of proofs and attacks can be attributed to multiple gaps between verified models and real-world protocols:

1. gaps between cryptographic models and standards,
2. gaps between standards and implementations,
3. gaps between APIs and application-level security, and
4. gaps between individually secure ciphersuites and their insecure composition.

We describe these gaps in more detail and explain how we try to bridge them in the miTLS (Microsoft-INRIA TLS) project.

miTLS: A Verified Reference Implementation of TLS

By 2008, TLS theory and practice had largely diverged. To relate high-level specifications and low-level implementations (gaps 1 and 2 above), a group of researchers at the Microsoft Research–INRIA joint center in Paris (including two authors of this article) decided to build a reference implementation of the TLS 1.0 standard (RFC 2246) in a style that enabled them to extract a formal model of the protocol directly from the code.[5] Using this approach, they ensured that the formal model was faithful to the standard and captured its low-level details. The model was then analyzed with a state-of-the-art protocol verifier, called ProVerif, to find both logical flaws in the protocol standard and implementation bugs in their code. Inasmuch as ProVerif didn't find any flaws, they obtained high assurance in the security of their code against a large class of attacks.

This reference implementation, later dubbed miTLS, was written in about 4,000 lines of F#, and the extracted symbolic models were among the largest to be automatically analyzed at the time, at the limits of verification technology. Symbolic tools like ProVerif are effective in automatically finding flaws without the need for user intervention, but they don't necessarily scale well to large models. Verifying their TLS implementation for one protocol version and one ciphersuite took 3.5 hours and 4.5 Gbytes of memory. Modeling other protocol modes was out of reach. Consequently, although they were able to find known attacks on early versions of SSL, they missed TLS renegotiation or triple handshake attacks that were discovered later, because their models didn't fully account for renegotiation.

As discussed earlier, a limitation of symbolic approaches is that they assume that the underlying cryptographic building blocks are perfect, and hence miss attacks. Some semiautomated tools, such as CryptoVerif, can analyze protocols in a more precise computational model of cryptography, but similarly don't scale up to large models. They applied CryptoVerif to core fragments of their TLS implementation but weren't able to analyze the full protocol using this tool. For example, they didn't model features like compression or the details of cipher block chaining, and hence they missed subsequent vulnerabilities like BEAST and CRIME.

For the next version of miTLS, we wanted to use a proof technique that could handle multiple versions and features of the protocol simultaneously and would rely on standard computational assumptions for the underlying cryptographic constructions.[6] To this end, we switched to a verification method based on refinement types (to be explained shortly), originally designed for symbolic protocol analysis by Karthikeyan Bhargavan and his colleagues,[7] and then extended to modular computational proofs by Cédric Fournet and his colleagues.[8]

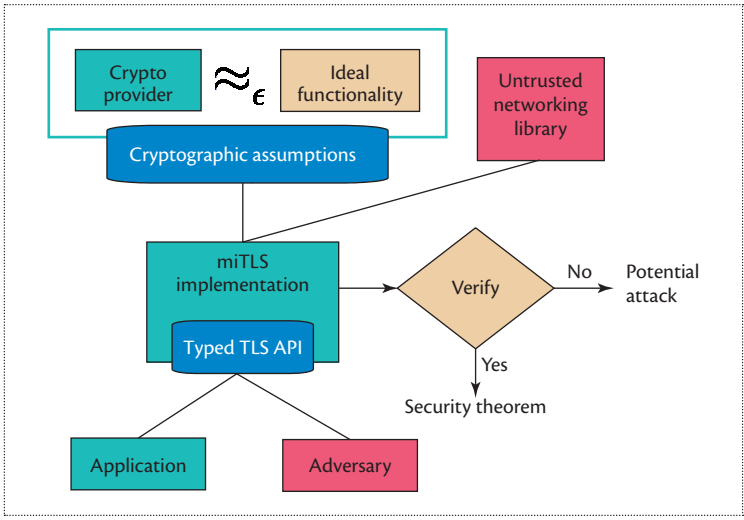

Figure 1. miTLS verification architecture. We verify that the miTLS implementation satisfies the security goals specified in its typed TLS API. Verification relies on cryptographic assumptions expressed as the types of an ideal functionality. If verification succeeds, miTLS is secure, as long as the crypto provider is indistinguishable (≈) from its ideal functionality.

Refinement types let programmers annotate each function with logical formulas. These annotations can capture program invariants, cryptographic assumptions, protocol events, and many security guarantees. To verify that a program meets its type annotations, the developer runs a type checker that automatically verifies the program with the aid of the Z3 (github.com/Z3Prover/z3) solver to discharge logical proof obligations. Crucially, type-checking is compositional in the sense that each function can be independently verified, assuming that all previous functions also meet their type annotations. Consequently, the time for type-checking a large program is more-or-less linear in its size and can be controlled by writing additional intermediate annotations.

The miTLS implementation currently supports TLS 1.0, 1.1, and 1.2, with multiple handshake and record modes. It also fully supports session resumption and renegotiation. The code is written in about 5,000 LOC and is split into a sequence of modules, each of which is equipped with a refinement-type interface. Figure 1 depicts the verification approach. For modules containing protocol code, the interface represents the target security goals we want to verify. For modules implementing cryptographic primitives, the interface represents the primitive's idealized functionality, according to some standard cryptographic security assumption.

miTLS's top-level security guarantees are stated in terms of a secure channel interface presented by TLS to the application. This interface guarantees that

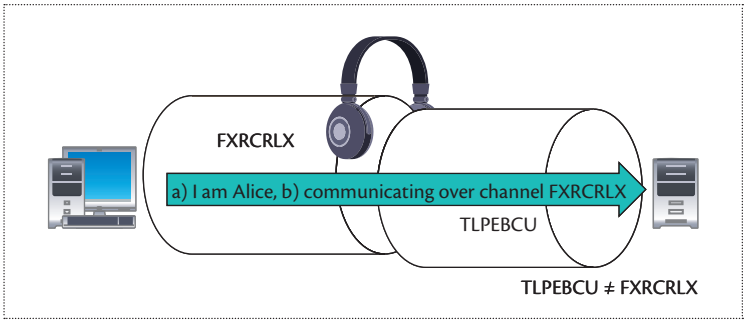

Figure 2. A compound authentication protocol over TLS. Alice inadvertently establishes a secure channel with Eve. If Alice authenticates within the channel (a), Eve can forward authentication messages within her own channel with Bob and impersonate Alice. Eve acts as a man in the middle (MITM) and observes communications Bob deems private. The attack can be prevented if Alice attaches her unique channel identifiers FXRCRLX to the authentication. Bob compares identifiers, detects the MITM, and aborts.

application data sent on a connection between a miTLS client and a miTLS server is kept confidential, as long as the connection uses strong cryptographic algorithms and the long-term private keys of the two peers are unknown to the attacker. Moreover, the interface guarantees that the stream of application data received at one end is a prefix of the stream sent by the other. The security proof relies on type-checking each module, after applying a series of game-based transformations on the core cryptographic modules to replace the concrete algorithms by their ideal functionality. Using this approach, we can verify the full miTLS implementation, module by module, under precise computational security assumptions. The total time for verification is under 20 minutes.

Although it's valuable to have a security theorem for a reference implementation of TLS 1.2, the impact of miTLS is perhaps better evaluated in terms of the parts of the protocol design we were unable to prove, or where we had to make special cryptographic assumptions. These corner cases resulted in the discovery of weaknesses in the protocol and attacks on its real-world usage.

API and Its Security Goals

Many problems stem from a mismatch between the security properties expected by applications using TLS and the actual guarantees provided by TLS (gap 3). The TLS standard doesn't specify an API, so each implementation is free to implement its own. Application developers are expected to understand these APIs in detail and to use them correctly to achieve their security goals. For example, some TLS libraries expect applications to validate the certificate presented by the server, and thus developers who wrongly assume that the library will do it for them become vulnerable to server impersonation attacks. More generally, many attacks appear when building application-level authentication on top of TLS.

Consider an application that uses TLS to establish a secure channel where the client is initially unauthenticated. The application then runs an authentication protocol on top of TLS that lets the user present a credential to the server. In this setting, the client expects that its use of TLS guarantees that the credential will be presented only at the target server, and a server that receives the credential over TLS might expect that the user intended to authenticate to it. However, as demonstrated by the attack outlined in Figure 2, these expectations are ill placed.

We follow cryptographic tradition and refer to the client as Alice, the server as Bob, and the attacker as Eve. If Alice is willing to use the same credential (say, an X.509 certificate) with both Bob and Eve, then Eve can impersonate Alice at Bob by forwarding Alice's credential (say, her signature over some authentication message) over her own channel with Bob. Even if Alice is careful and uses her credential only with Bob, a sophisticated attacker might impersonate Bob to mount a man-in-the-middle (MITM) attack by operating a phishing website, obtaining misissued certificates, or compromising the server key. Such credential-forwarding attacks can be prevented if Alice authenticates not only herself but also her channel, for instance, by signing a unique identifier extracted from the TLS connection. Then, if Bob compares these channel identifiers, he can detect the attack.

Credential forwarding attacks and their countermeasures have appeared multiple times in TLS applications. They were first discussed in the context of tunneled compound authentication protocols for network access. They then reappeared in the context of user-authenticated TLS renegotiation as commonly used on the Web. In response to these attacks, a variety of channel identifiers were defined for TLS and exposed in the APIs of various implementations. Compound authentication protocols used the TLS session key (called *master secret*) as an identifier for binding application-level credentials. TLS renegotiation used the protocol transcript of the previous handshake as a connection identifier.

We implemented these countermeasures in miTLS and tried to prove that applications using miTLS aren't vulnerable to credential forwarding, but we failed. Instead, we discovered several counterexamples. A malicious server can synchronize the TLS session keys on two different connections, one from Alice and one to Bob, so that the channel identifiers on both connections are the same, hence defeating the compound authentication countermeasure. Then by running a second TLS connection that uses session resumption, the server

can also synchronize the protocol transcripts on these connections, hence defeating the TLS renegotiation countermeasure. In fact, such channel synchronization attacks break all known credential forwarding protections over TLS by exploiting a misunderstanding of the TLS API; the protocol doesn't guarantee unique channel identifiers.

This class of channel synchronization attacks was called the *triple handshake*,[9] because it requires a sequence of up to three runs of TLS before the attack succeeds against TLS client authentication. Although it has been present in the TLS protocol since SSL3, it escaped previous analyses because researchers didn't consider TLS connection sequences and didn't model credential forwarding as a threat. In response to these attacks, we helped the TLS working group standardize a new protocol-level fix called the *extended master secret* that systematically protects all compound authentication protocols.

Implementing Negotiation

In addition to designing an API, a second major challenge for a TLS implementation is that it needs to handle a variety of protocol versions, extensions, authentication modes, and ciphersuites simultaneously. Although the TLS standard describes each mode in isolation, it doesn't always specify how an implementation should compose them (gap 4). In particular, the protocol state machine is left unspecified, and each implementation can design its own. In miTLS, we define and verify our own state machine. Our type-based proofs rely on careful invariants that require that the current protocol state is consistent with the desired protocol mode and that the transcripts and signature formats for different modes are disjoint. Considering the effort that was required to prove our own state machine correct, we tested other implementations to see if they implemented the TLS standard correctly. To our surprise, many of them failed this test, resulting in subtle attacks.[10]

Some implementations failed to correctly implement the composition of the handshake and record protocols and allowed application data to be sent unencrypted before the handshake was complete. Other implementations failed to correctly compose regular RSA ciphersuites with export RSA ciphersuites, allowing a downgrade attack, called FREAK, whereby an MITM attacker could fool a TLS client into accepting export-grade 512-bit RSA keys even though it wanted to use regular RSA. In all, by testing other open source TLS libraries against miTLS, we found dozens of state machine bugs across all major TLS implementations, including four that could be exploited for real-world attacks.

Another attack on TLS negotiation, found by a large group of researchers including one of the authors of this article, relies on a protocol flaw rather than an implementation bug.[11] In Logjam, the server supports both regular Diffie-Hellman (DH) groups as well as export-grade 512-bit DH groups. The client doesn't support export-grade DH, but it allows the server to pick the group. As Figure 3 depicts, this situation leads to an MITM attack. The attacker tampers with the protocol messages to fool the server into thinking the client supports only export-grade DH. So, the server sends the export-grade group to the client who thinks this is the server's regular group and accepts it. This is sometimes called a *cross-protocol attack* because it involves confusion between two different protocols (DH and export DH). It's enabled by a TLS protocol flaw: the server's signature format for export DH ciphersuites is indistinguishable from its signature for regular DH. Hence, the attacker can successfully downgrade the connection to use export DH even though the client doesn't realize it. To complete the attack, the attacker must still solve the discrete log problem for the export DH group, which is well within reach of modern processing power.

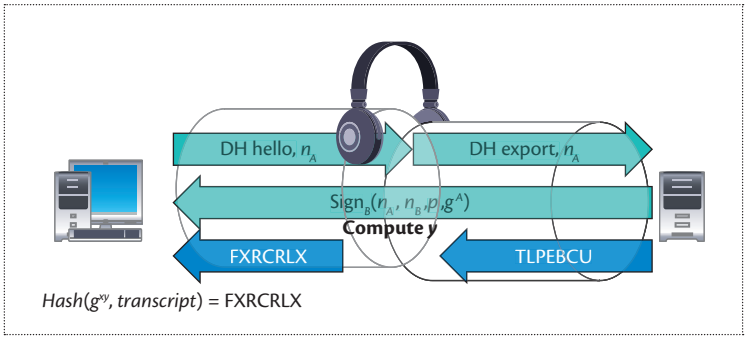

Figure 3. The Logjam attack. Alice wants to communicate with Bob. Eve tricks Bob into starting a Diffie-Hellman (DH) export protocol. Eve forwards the signature of Bob to Alice to get her to use a small prime. Then Eve computes the discrete logarithm y to compute g^{xy}. Using this secret, she can compute the channel keys to decrypt Alice's messages.

Toward TLS 1.3

When SSL was first designed, there was a real enthusiasm and sense of purpose to deploy practical cryptographic protocols. Often one and the same person worked on and understood both the cryptography and the implementation—to the extent possible at the time. Since then, advances in cryptographic theory and analysis have greatly improved our understanding of when protocols achieve their security goals and when they fail to do so. However, typically this analysis is performed either on toy protocols or in retrospect on partial aspects of a protocol specification. Implementers still primarily follow a fix-attack-fix cycle. This cycle, however, only gets worse as protocols grow in complexity. We believe that the only way forward is

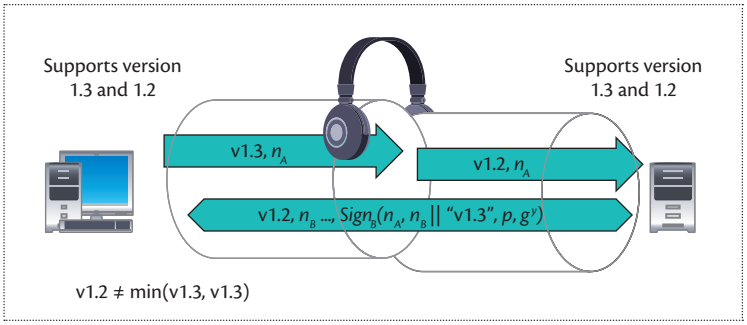

Figure 4. The version downgrade countermeasure. Alice tries to establish a TLS 1.3 connection with Bob, but Eve changes the maximum client version to TLS 1.2 to attempt a version downgrade. The attack can be prevented if Bob signs his maximum version. Alice compares Bob's maximum version with its chosen version to detect Eve's tampering and aborts.

through an active collaboration between theoreticians and practitioners.

A promising development in this direction is the standardization effort behind the upcoming TLS 1.3 protocol, which fixes many weaknesses in TLS 1.2 and, at the same time, promises improved performance. From the early stages of its design, the TLS working group has invited and encouraged the participation of academic researchers, who have responded in significant numbers. Not only was the design of the cryptographic core of TLS 1.3 strongly influenced by the OPTLS protocol by Hugo Krawczyk and Hoeteck Wee,[12] but we now have multiple published security proofs for different draft versions of the protocol even before it has been standardized. Such careful cryptographic analysis for a new standard is unprecedented at the IETF. As a result of this process, many attacks and weaknesses were detected and removed from early drafts, resulting in a simpler and more secure protocol.

Our main contribution to the standardization effort is a new version of miTLS that implements TLS 1.3 but also supports older versions for backward compatibility. Because mainstream TLS implementations will continue to support such older versions for the foreseeable future, we were especially concerned with the potential for version downgrade attacks that might nullify the security advantages of TLS 1.3.

TLS 1.3 signs all exchanged messages to prevent MITM attacks like Logjam that rely on tampering with handshake messages for downgrade attacks. However, we discovered that by downgrading the protocol version to TLS 1.2, the attacker can force the server to use the weaker TLS 1.2 signature that doesn't cover all messages, hence reenabling such tampering attacks. The problem is that, in older versions of TLS, clients can't verify the maximum supported server version until the end of the protocol, by which time it's too late.

That this downgrade attack went unnoticed until TLS 1.3 draft 10 illustrates the many intricacies and pitfalls of practical protocol security. Once detected and brought to the IETF's attention, we helped develop a verified countermeasure, depicted in Figure 4, that is peculiar but simple: shorten the server nonce, which is signed in TLS 1.2, and use some of its bytes to encode the server's highest supported version number.

The miTLS approach necessarily involves multidisciplinary teams of cryptography, programming semantics, tooling and verification experts, and generalists knowledgeable of real-world security concerns and system performance. We require implementations to be written in a programming language with well-defined formal semantics so that protocol properties devised by theorists can be verified using sound-automated tools on code codeveloped with practitioners.

Even verified implementations have to rely on cryptographic assumptions, the accuracy of the security model, and the correctness of proofs and verification tools. To ensure that our modeling assumptions catch concrete attacks, we advocate a comprehensive penetration testing regime that uses the miTLS code base to find and implement attacks on miTLS and other TLS implementations. Such attacks can be on cryptographic primitives, on the TLS protocol level, on the HTTPS ecosystem, and even against the soundness of our verification tools. Our goal is to use a combination of verification and testing to span and evaluate all four of these levels to reduce the trusted computing base for TLS applications.

Verification alone is not enough to ensure that a TLS implementation will be widely used. Real-world implementations must be performant. A key challenge for future work on miTLS is to extend our verification techniques so that they can handle the programming idioms used in high-performance code. As our tools improve, we anticipate that the feature and performance gap between verified and unverified protocol implementations will vanish. ■

References

1. S. Vaudenay, "Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS…," *Proc. Int'l Conf. Theory and Applications of Cryptographic Techniques* (EUROCRYPT 02), 2002, pp. 534–546.
2. M. Abadi and R. Needham, "Prudent Engineering Practice for Cryptographic Protocols," *IEEE Trans. Software Engineering*, vol. 22, no. 1, 1996, pp. 2–15.
3. D. Wagner and B. Schneier, "Analysis of the SSL 3.0 Protocol," *Proc. USENIX Workshop Electronic Commerce*, 1996, pp. 29–40.

4. J.P. Degabriele, K. Paterson, and G. Watson, "Provable Security in the Real World," *IEEE Security & Privacy*, vol. 9, no. 3, 2011, pp. 33–41.
5. K. Bhargavan et al., "Cryptographically Verified Implementations for TLS," *Proc. 15th ACM Conf. Computer and Communications Security* (CCS 08), 2008, pp. 459–468.
6. K. Bhargavan et al., "Implementing TLS with Verified Cryptographic Security," *IEEE Symp. Security and Privacy* (SP 13), 2013, pp. 445–459.
7. K. Bhargavan, C. Fournet, and A.D. Gordon, "Modular Verification of Security Protocol Code by Typing," *Proc. 37th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 10), 2010, pp. 445–456.
8. C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular Code-Based Cryptographic Verification," *Proc. ACM Conf. Computer and Communications Security* (CCS 11), 2011, pp. 341–350.
9. K. Bhargavan et al., "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS," *IEEE Symp. Security and Privacy* (SP 14), 2014, pp. 98–113.
10. B. Beurdouche et al., "A Messy State of the Union: Taming the Composite State Machines of TLS," *IEEE Symp. Security and Privacy* (SP 15), 2015, pp. 535–552.
11. D. Adrian et al., "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," *Proc. ACM Conf. Computer and Communications Security* (CCS 15), 2015, pp. 5–17.
12. H. Krawczyk and H. Wee, "The OPTLS Protocol and TLS 1.3," *IEEE European Symp. Security and Privacy* (EuroS&P 16), 2016, pp. 81–96.
13. K. Bhargavan et al., "Downgrade Resilience in Key-Exchange Protocols," *IEEE Symp. Security and Privacy* (SP 16), 2016, pp. 506–525.

Karthikeyan Bhargavan is a researcher at INRIA, the French national lab for computer science, where he leads a team called Prosecco ("programming securely with cryptography"), and is the principal investigator of an ERC consolidator grant CIRCUS on provably secure implementations of cryptographic Web applications. Bhargavan was trained at IIT New Delhi and received a PhD in computer science from the University of Pennsylvania. Contact him at karthikeyan.bhargavan@inria.fr.

Cédric Fournet leads the Constructive Security group at the Microsoft Research lab in Cambridge, UK. His research interests include security, privacy, cryptography, programming languages, and formal verification, and he's currently working on a verified TLS/HTTPS protocol stack and techniques for outsourcing computations with strong security and privacy guarantees. Fournet received a PhD in computer science from INRIA. Contact him at fournet@microsoft.com.

Markulf Kohlweiss is a researcher in the Programming Principles and Tools group at Microsoft Research Cambridge. His research focus is on privacy-enhancing cryptography and formal reasoning about cryptographic protocols. Kohlweiss received a PhD in cryptography from the COSIC (Computer Security and Industrial Cryptography) group at the KU Leuven. Contact him at markulf@microsoft.com.