Automated Proof and Flaw-Finding Tools in Cryptography

Graham Steel | Cryptosense

One of the few things cryptographers can agree on is that cryptography is a complex subject—so complex that even after expert peer review, flaws are regularly found in security proofs, APIs, protocols, and protocol implementations. Computer scientists have long hoped that computers could do the work of checking hardware and software for flaws, but this idea—called formal analysis—has only recently been successfully implemented in cryptography.

Unlike conventional software verification, cryptographic analysis requires verifiers to account for all possible actions of a malicious intruder trying to break into the system. To facilitate this task, researchers have turned to algebraic modeling as a way to produce abstract models of crypto protocols.

Dolev and Yao’s Verification Model
In 1983, Daniel Dolev and Andrew C. Yao proposed modeling cryptographic protocols and a hypothetical attacker in an abstract way to allow verification of the protocols’ security.1 By modeling bitstrings as terms in abstract algebra, cryptographic algorithms as functions of those terms, and an intruder as a set of deduction rules on the abstract algebra, Dolev and Yao (DY) were able to verify simple protocols in their model.

Formal tools such as model checkers and theorem provers that matured through the 1990s all used the DY protocol verification model. By the early 2000s, several purpose-built tools were capable of verifying abstract DY models of large protocols for unbounded numbers of protocol sessions. However, the DY model is usually an approximation rather than an abstraction of the protocol implementation. This means that attacks found in a DY model don’t always correspond to real attacks on an implemented protocol and—more worryingly—that an absence of attacks in the DY model doesn’t generally imply absence of attacks on the real protocols.

Formal Proofs in the Standard Model
Proofs in the standard model can be strong but are notoriously difficult to get right. One approach is to formalize the proof using a proof assistant, a piece of software that mechanizes logical reasoning with a very high degree of confidence of correctness. This high confidence comes from the fact that the trusted code base—the part of the software we have to believe is correct to trust the proofs—is very small and has been tested extensively.

Successful approaches using this method in other fields include using the Coq proof assistant to verify proofs of the four-color theorem (http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf) and, more recently, using Isabelle and...
HOL Light proof assistants to verify Johannes Kepler’s conjecture (http://arxiv.org/abs/1501.02155). The Computer-Aided Cryptography group at the IMDEA Software Institute and INRIA Sophia Antipolis developed the EasyCrypt proof assistant and used it to verify security proofs for several well-known cryptographic primitives (such as OAEP^{2}) and protocols (such as NAXOS^{3}). EasyCrypt is technically impressive but requires significant user expertise and effort.

Although proving the security of a protocol specification is useful, security flaws are often introduced at the implementation level. Several research groups have tackled this problem over the past 10 years. One idea is to take a formally verified protocol specification and compile it into an implementation that’s guaranteed to be secure.\(^{4}\) Another approach is to prove an implementation’s security in a high-level language that’s amenable to formal verification, enabling the specification of rich properties that translate to cryptographic security guarantees.

An example of this approach is the miTLS (www.mitls.org) software for the widely used Web protocol SSL/TLS. Developed by Karthikeyan Bhargavan’s team at INRIA in the functional programming language F#, this software includes a full implementation of the standard protocol including multiple cipher suites, resumption, and renegotiation of sessions. The use of the memory-safe F# language guarantees an absence of the kinds of buffer-overflow bugs that led to the widely publicized Heartbleed vulnerability discovered in 2014 (http://heartbleed.com).

Bhargavan’s team then used F7 programming language to specify the security of TLS and F7’s refinement type checker and verify that the implementation instantiates the specification. Their work shows how cryptographic security guarantees, which are typically seen only for low-level primitives and protocols, can be scaled up to a real implementation of a complex protocol.

While completing the specification and implementation of their software, which requires a deep understanding of the protocol, the miTLS team discovered several exploitable flaws in the most widely used implementations (www.secure-resumption.com; www.smacktls.com). Other automated analysis approaches focus on finding vulnerabilities in real implementations without looking at the source code. Erik Poll’s group at the Institute for Computing and Information Sciences at Radboud University has recently shown how to determine a protocol’s state machine from its black-box implementation using the L∗ algorithm. Once a model has been created, it can be explored with a model checker, allowing the discovery of paths leading to bugs.

Comparing several implementations of the same protocol has also proved fruitful in finding flaws. Poll’s team applied this approach to chip and PIN cards and, more recently, to implementations of TLS. In doing so, they uncovered several anomalies and security bugs.\(^{5}\)

**Although proving the security of a protocol specification is useful, security flaws are often introduced at the implementation level.**

**API-Level Attacks**

When developing new enterprise applications that will use cryptography, developers typically use existing software or hardware implementations of cryptographic algorithms rather than creating their own. These algorithms are accessed via an API. For this reason, it’s also necessary to study cryptographic security at the API level to see whether, for example, a rogue application could extract cryptographic keys from a piece of secure cryptographic hardware by calling an unexpected sequence of commands.

Examples of formal analysis being used to search for attacks on APIs date back to the early 1990s,\(^{6}\) but the subject came to prominence in the early 2000s when a series of such attacks was discovered on the hardware security modules (HSMs) widely used in banking and other security-critical applications.\(^{7,8}\) These attacks were discovered by manual analysis or ad hoc tools, so several research groups wanted to see if the general API security problem could be tackled using formal tools similar to those used for DY-style protocols.

In 2010, Matteo Bortolozzo and his colleagues demonstrated Tookan—a tool capable of learning an implementation model of RSA PKCS#11, the most widely used standard for cryptographic hardware—by sending a call sequence and observing the error responses.\(^{9}\) After learning a model, Tookan queries a model checker to search for sequences of calls that could reveal the value of a sensitive cryptographic key. If such a sequence is found, Tookan converts the sequence of steps in the model back into a sequence of real PKCS#11 API calls and executes the attack on the hardware being tested. Using this methodology, Tookan was able to find key-recovery attacks on several commercially available PKCS#11 smart cards and HSMs.

Following industrial interest, a spin-off company of INRIA called Cryptosense now commercializes
the tool under the name Crypto-sense Analyzer. Many large banks and security agencies use it to audit cryptographic hardware and ensure that the crypto infrastructure is configured securely.

**Side-Channel Attacks**

Side-channel attacks are significant threats to real cryptography implementations. These attacks rely on the information leakage by power consumption, execution timing, electromagnetic or audio emissions, or the error messages given by a cryptographic application. New side channels are continually being discovered, and each typically has its own specific set of countermeasures. However, the idea of automatically verifying the effectiveness of a given side channel’s countermeasures has only recently begun to receive attention.

One defense against attacks that rely on variations in execution time is to implement cryptography in such a way that the execution time is constant regardless of the data or the value of the cryptographic key. Tools that can verify whether a piece of code will execute in constant time are beginning to emerge (https://github.com/agl/ctgrind); however, this analysis isn’t easy. Code that seems to be constant at the source code level might not always produce constant-time machine code. Recent work has looked directly at machine code. Many open challenges remain in verifying the absence of side-channel attacks.

Current trends such as cloud computing, smartphones, and the Internet of Things are creating a growing need for security that will be provided by cryptography. As a consequence, developers are now required to implement cryptography without necessarily developing the expertise needed to do it securely. It’s safe to say we’ll see more automated analysis technologies for all aspects of cryptography in the years to come.

**References**


Graham Steel is CEO and cofounder of Cryptosense, a spin-off of INRIA that produces software for security analysis of cryptographic systems. Contact him at graham.steel@cryptosense.com.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.