Short Paper: Modular Black-box Runtime Verification of Security Protocols

Kevin Morio
CISPA Helmholtz Center for Information Security
Saarbrücken Graduate School of Computer Science

Dennis Jackson
Department of Computer Science
ETH Zürich

Marco Vassena
CISPA Helmholtz Center for Information Security

Robert Künnemann
CISPA Helmholtz Center for Information Security

ABSTRACT

Verification techniques have been applied to the *design* of secure protocols for decades. However, relatively few efforts have been made to ensure that verified designs are also *implemented* securely. Static code verification techniques offer one way to bridge the verification gap between design and implementation, but require substantial expertise and manual labor to realize in practice. In this short paper, we propose black-box runtime verification as an alternative approach to extend the security guarantees of protocol designs to their implementations. Instead of instrumenting the complete protocol implementation, our approach only requires instrumenting common cryptographic libraries and network interfaces with a runtime monitor that is *automatically synthesized* from the protocol specification. This lightweight technique allows the effort for instrumentation to be shared among different protocols and ensures security with presumably minimal performance overhead.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; *Security protocols*; Software security engineering.

KEYWORDS

runtime monitoring, protocol verification

ACM Reference Format:
Kevin Morio, Dennis Jackson, Marco Vassena, and Robert Künnemann. 2020. Short Paper: Modular Black-box Runtime Verification of Security Protocols. In *15th Workshop on Programming Languages and Analysis for Security (PLAS'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3411506.3417596

1 INTRODUCTION

Security protocols such as TLS [23], Signal [26] and Wireguard [12] are the basis of secure communication over the Internet. Such protocols aim to provide an abstract communication channel with specific security properties, e.g., authentication, confidentiality, and privacy, to higher-level applications. The design of these protocols is a challenging endeavor that culminates in one or more proofs that justify their security properties. These proofs are often carried out manually, but due to their complexity, mistakes are regularly discovered [5, 11, 20]. This has motivated the development of automatic provers and proof assistants such as Tamarin [19], ProVerif [8] and EasyCrypt [10]. Such tools accept a formal description of the protocol and the expected security property and establish whether the protocol satisfies the property, or assist humans in creating a machine-checked proof.

However, both manual and tool-assisted proof techniques only establish a correspondence between the design of a protocol and its security properties, they do not cover the *implementation* of the design. This leaves a gap, where protocols may be securely designed, but insecurely implemented. Several works have attempted to bridge this gap by using *static verification* techniques to formally verify that a particular implementation adheres to its high-level design. These efforts employ various white-box techniques, including separation logics [21], program refinement [25], and verification and compilation toolchains [22]. Whilst these techniques have delivered promising results and are the subject of ongoing research, they do require considerable expertise and a significant manual effort to realize. In this short paper, we propose an alternative approach based on *runtime verification.* Intuitively, we verify that protocol implementations follow their specification at runtime in a black-box fashion, by instrumenting the underlying cryptographic libraries and network interfaces. In contrast with the manual and ad-hoc effort required by static verification techniques, this is a once and for all effort that is not specific to any protocol and can be shared by different protocols. Further, whilst we anticipate a minor performance impact since the monitor executes together with the protocol itself, we believe this approach can effectively extend the security properties of the protocol specification to its implementation. Our runtime monitor promises strong guarantees that protocol implementations match their *verified* specifications and supports complex protocols with looping and branching control flows. Since the monitor requires only instrumenting a small part of an implementation, we expect minimal performance overhead.

2 SOLUTION OVERVIEW

Our approach is based on online runtime monitoring, a well-established lightweight verification technique to establish system correctness and reliability [17]. At a high level, this approach consists of executing the protocol implementation together with a runtime monitor that ensures that the execution adheres to the protocol specifications at runtime. However, our monitor does not rely on the protocol code to detect violations. Instead, our approach is black-box and only intercepts calls to cryptographic libraries and network interfaces to catch security bugs in the protocol implementation. Concretely, the monitor detects when the code execution is about to diverge from its specification and acts accordingly to prevent security breaches. We elucidate possible reaction strategies in Section 3.

Protocol Specifications Formal protocol specifications are usually specified using process calculi [1, 8] or multiset rewriting [19]. The transition rules in these systems are labeled with actions that model the protocol receiving and sending (encrypted) messages over the network. These messages carry *symbolic terms*, which represent inter alia plaintexts and ciphertexts and prescribe how cryptographic primitives must be composed by the protocol in order to provide secure communication. In these formalisms, the actions and the messages generated by the rules form a *symbolic trace* that captures the protocol execution. Then, the security guarantees of the protocol, e.g., weak secrecy [7] and authentication [18, 27], are defined as a *trace property*. The security of the protocol is then established by showing that the transition system always generates *good traces*.

Runtime Monitor We ensure that protocol implementations follow their specification by automatically synthesizing runtime monitors from labeled transition systems describing the protocol in question. The fact that the security of a protocol execution depends only on the messages exchanged and their content offers a way to enforce security in a lightweight black-box fashion. Intuitively, to establish whether the protocol implementation complies with the specification, the monitor only needs to track specific operations, i.e., calls to cryptographic libraries and network interfaces. The monitor intercepts these operations while the protocol implementation is running and builds a trace of actions containing *concrete values*, e.g., ciphertext bitstrings. Then, the concrete trace is matched by the monitor against the expected *symbolic trace* generated by the labeled transition system in order to detect inconsistencies. Importantly, the monitor fulfills this last step my maintaining a mapping between concrete values and their corresponding symbolic terms, which allows interpreting bitstrings as symbolic terms and vice versa. Notice that the runtime monitor is run per protocol party (agent), which may be engaged in concurrent sessions and acting in more than one role. As a result, the traces observed by the monitor may be spread over multiple sessions or multiple roles, thus complicating the synthesis of the monitor.

QEA Our runtime monitor is based on the notion of *quantified event automata* (QEA), a recently proposed formalism for describing expressive and efficient runtime monitors [4]. Event automata are non-deterministic finite-state machines that specify *parametric* trace properties. Intuitively, the transitions of the automata are annotated with events that contain *variables*, which get instantiated with concrete values as the automata consumes the input trace. QEAs boost the expressiveness of event automata by additionally allowing these variables to be *quantified*. Quantified variables spawn multiple instances of the event automata, each binding quantified variables to concrete values. As a result, a QEA accepts a trace as long as subsequences of these trace can be individually accepted by the instantiated event automata. This formalism captures precisely the interleaving semantics of protocol sessions running in parallel, which can then be efficiently monitored by the automata.

Mapping Our monitor bridges the gap between symbolic and concrete values by maintaining two variable binding maps, one to symbolic terms and one to bitstrings. These maps help the monitor to refine the knowledge of concrete values during the program execution and interpret them correctly as symbolic terms. For example, suppose that a protocol is expected to receive an encrypted message, specified by a symbolic term $enc(k,m)$. When the protocol receives this message from the network interface, the QEA monitor sees the bitstring corresponding to the ciphertext, but does not know that it is a ciphertext yet. The term map thus points to a variable, e.g., t. However, once the message is successfully decrypted through a call to the appropriate cryptographic API, the term mapping can be refined, updating t to $enc(k,m')$ where k is the term representation of the key parameter, and m' the term representation of the function output. Moreover, if the bitstring returned by the function corresponds to the bitstring representation of m and m is an atomic data type (e.g., payload), m and m' will be unified to the same term.

Translation Overview We illustrate how to automatically synthesize a runtime monitor from a protocol specification by outlining a *translation procedure* from process calculi into QEA. We give an overview about the translation approach for the five most essential features of process calculi: parallel execution, replication, input, output and choice of fresh names. We translate parallel process composition by translating each process into a QEA and providing *silent* ϵ-transitions into each of them, as exemplified in Fig. 2. Typically, these processes are under replication, i.e., they can be repeated arbitrarily often. To this end, we add a transition from each final node of the translated QEA back to the initial node and quantify over the variables in the process. This quantification requires that variables are uniquely named, such that each variable occurs in exactly one input construct of the process. Fig. 2 depicts a simple process with input, output and fresh name generation. Both inputs and outputs are translated into labeled transitions; input transitions are characterized by the variable they bind, while output events by the terms they contain. To treat fresh names, we need to extend QEAs with uniqueness quantification ($\exists!$), e.g., if the process in Fig. 2 was under replication, it would be quantified $\forall x.\ \exists!\, r$, meaning that there exists a unique name r for each binding of x.

Formal Guarantees and Evaluation The key property that we intend to prove for our runtime monitor is *soundness*. Intuitively, soundness guarantees that the concrete traces accepted by the monitor correspond to a possible symbolic trace generated by the protocol specification, i.e., the implementation cannot silently deviate from its specification. Furthermore, soundness allows to automatically extend the security properties of the specification to the monitored implementation. In particular, if a protocol specification is shown to satisfies a given trace property (e.g., by using Tamarin or ProVerif), then it is also satisfied by the monitored implementation. Although runtime monitoring would not in general preserve hyperproperties (i.e., properties of sets of traces),[1] several properties of interests for protocols are simple trace properties (e.g., weak secrecy and injective

[1]This could change in the future [24], but recent advances in runtime monitoring could enable efficient monitoring of hyperproperties [14–16].

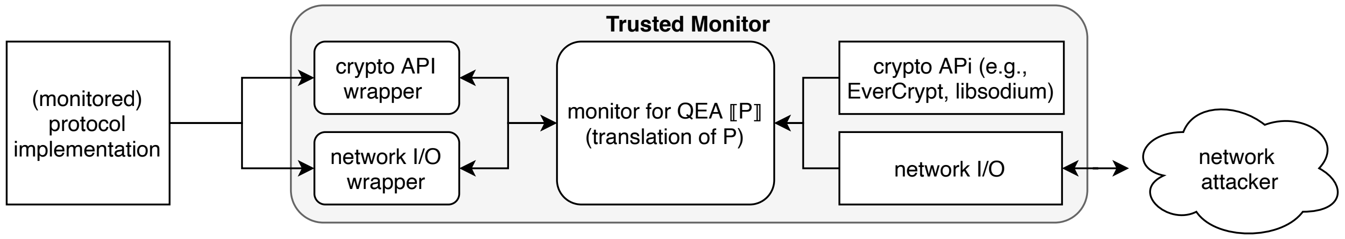
Trusted Monitor
(monitored) protocol implementation
crypto API wrapper
network I/O wrapper
monitor for QEA ⟦P⟧ (translation of P)
crypto APi (e.g., EverCrypt, libsodium)
network I/O
network attacker

Figure 1: Architecture of the monitor. P is the protocol specification and $[\![P]\!]$ is the QEA monitor synthesized from P.

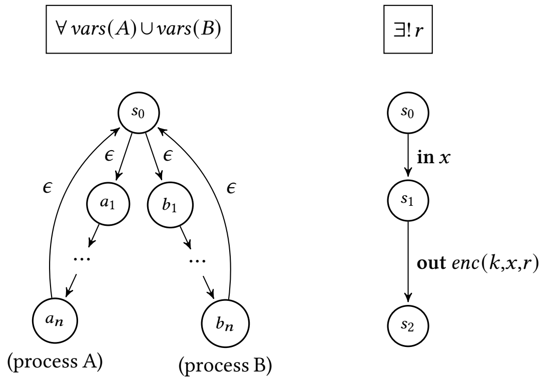

Figure 2: Left: Two replicated processes in parallel. Right: Input, output and generation of fresh name r.

authentication). Moreover, some interesting hyperproperties, for instance those based on simulation, are closed under trace inclusion and would hence be preserved.

In this work, we focus on honest-but-buggy protocol implementations. We do not consider malicious implementations which have been deliberately subverted to evade detection of the monitor, e.g., through covert- and side-channels. We also assume that the implementation of the underlying cryptographic primitives is correct and secure, for example through existing verification techniques [3, 6, 9].

3 CHALLENGES AND LIMITATIONS

Recovery Our initial focus is on *soundness*, i.e., we anticipate that the monitor will have to be conservative and sometimes flag secure executions as potentially insecure. We plan to evaluate the strictness of the monitor on real world protocols and investigate which trade-offs are necessary to reduce the rate of false positives in practice. Furthermore, even with a low false-positive rate, there may be better ways to avoid a (potential) security violation than aborting the protocol. We intend to investigate alternative reaction strategies for our monitors [13]. Besides simply logging violations (e.g., while the protocol implementation is under development), the monitor could also eliminate certain violations by actively modifying the protocol execution. These active reactions include: (1) *suppressing* unexpected output messages, at the price of potentially desynchronizing the implementation, and (2) *correcting* invalid output messages (e.g., a message encrypted with the wrong cryptographic key).

Conversion between values and symbolic terms The monitor must maintain a mapping between observed values and their symbolic representation in order to detect violations. This will likely require integration with network parsing and serialization libraries in order to interpret the structure of network messages. Further, by observing calls to cryptographic APIs, the monitor could deduce the correct symbolic representation for a particular value contained in a message. Intuitively, many of these values, e.g., cryptographic keys, hashes and ciphertexts, are guaranteed not to collide (with overwhelming probability), which would enable an efficient implementation in practice.

Session Resumption Many real-world protocol implementations permit interrupting and resuming protocol sessions over time by saving relevant session data in a persistent storage. Crucially, the code responsible for saving and restoring sessions is inherently complicated and therefore prone to bugs, which could weaken, or even break the security of the protocol. For example, a recently discovered session resumption bug in GnuTLS allowed network attackers to compromise the confidentiality of TLS sessions [2]. Hence, handling session resumptions is a key goal for our black-box monitoring technique. This means that the runtime monitor may need to persist state to inform future monitor instances about past actions taken by the protocol. However, extending the monitor to handle interrupted and resumed sessions without inspecting the protocol code is challenging. We believe that handling session resumptions will probably require the instrumentation of other interfaces besides the network API, e.g., the database management system and the file system API, used by the implementation to store session data.

Enriching Abstract Symbolic Models In protocol specification languages, the symbolic abstraction of counters and timestamps is typically left to the discretion of the modeler. However, protocol implementation must use concrete representations and ensure that desired properties such as uniqueness and monotonicity hold. Although it may be necessary to enrich some symbolic models to provide enforceable security properties, we hope that we can automate this process for many protocols. For example, we can treat timestamps in much the same way as cryptographic values by monitoring calls to time and date APIs.

Atomicity of Operations In the symbolic model, the actions taken by a protocol in response to inputs are typically atomic, that is, they either all occur or none of them occur. However, in practice, the protocol may perform such actions non-atomically, in a different order, and even omit some actions altogether. Some re-orderings may be innocuous, e.g., constructing two independent ciphertexts, but others may be catastrophic, for example, transmitting a message but not updating the state of the protocol. Clearly, the runtime monitor must strike a balance between over zealously enforcing the protocol specifications and missing security relevant violations. It may be possible for the monitor to automatically enforce that events appear to happen atomically from the perspective of a network observer and this could offer the correct balance.

REFERENCES

[1] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *28th ACM Symp. on Principles of Programming Languages (POPL'01)*. ACM, 104–115.

[2] GnuTLS Airtower. 2020. *CVE-2020-13777: TLS 1.3 session resumption works without master key, allowing MITM*. https://gitlab.com/gnutls/gnutls/-/issues/1011 (Accessed June 26th, 2020).

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 53–70.

[4] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. 2012. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM 2012: Formal Methods*, Dimitra Giannakopoulou and Dominique Méry (Eds.). Lecture Notes in Computer Science, Vol. 7436. Springer Berlin Heidelberg, 68–84. https://doi.org/10.1007/978-3-642-32759-9_9

[5] Daniel J. Bernstein. 2015. Multi-user Schnorr security, revisited. *IACR Cryptology ePrint Archive* 2015 (2015), 996.

[6] Karthikeyan Bhargavan, Benjamin Beurdouche, Jean-Karim Zinzindohoué, and Jonathan Protzenko. 2017. HACL*: A Verified Modern Cryptographic Library. *ACM CCS* (September 2017). https://www.microsoft.com/en-us/research/publication/hacl-a-verified-modern-cryptographic-library/

[7] Bruno Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th Computer Security Foundations Workshop (CSFW'01)*. IEEE Comp. Soc., 82–96.

[8] Bruno Blanchet. 2013. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *Foundations of Security Analysis and Design VII*. Springer, 54–87.

[9] Barry Bond, Chris Hawblitzel, Manos Kapritsos, Rustan Leino, Jay Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*. USENIX. https://www.microsoft.com/en-us/research/publication/vale-verifying-high-performance-cryptographic-assembly-code/ Distinguished Paper Award.

[10] Ran Canetti, Alley Stoughton, and Mayank Varia. 2019. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 167–16716.

[11] Kim-Kwang Raymond Choo, Colin Boyd, and Yvonne Hitchcock. 2005. Errors in Computational Complexity Proofs for Protocols. In *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings (Lecture Notes in Computer Science)*, Bimal K. Roy (Ed.), Vol. 3788. Springer, 624–643. https://doi.org/10.1007/11593447_34

[12] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/

[13] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *Runtime Verification*, Christian Colombo and Martin Leucker (Eds.). Vol. 11237. Springer International Publishing, Cham, 241–262. https://doi.org/10.1007/978-3-030-03769-7_14

[14] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2020. Efficient monitoring of hyperproperties using prefix trees. *International Journal on Software Tools for Technology Transfer* (02 2020). https://doi.org/10.1007/s10009-020-00552-5

[15] Christopher Hahn. 2019. Algorithms for Monitoring Hyperproperties. In *Runtime Verification*, Bernd Finkbeiner and Leonardo Mariani (Eds.). Springer International Publishing, Cham, 70–90.

[16] Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Constraint-Based Monitoring of Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 115–131.

[17] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.

[18] G. Lowe. 1997. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*. 31–43.

[19] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*. Springer, 696–701.

[20] Alfred Menezes. 2007. Another look at HMQV. *J. Mathematical Cryptology* 1, 1 (2007), 47–64. https://doi.org/10.1515/JMC.2007.004

[21] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS)*, B. Jobstmann and K. R. M. Leino (Eds.), Vol. 9583. Springer-Verlag, 41–62.

[22] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 2019. Formally verified cryptographic web applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1256–1274.

[23] E. Rescorla. 2018. *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3*. https://tools.ietf.org/html/rfc8446 (Accessed May 29th, 2020).

[24] Shubham Sahai, Pramod Subramanyan, and Rohit Sinha. 2020. Verification of Quantitative Hyperproperties Using Trace Enumeration Relations. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 201–224.

[25] Christoph Sprenger and David Basin. 2018. Refining Security Protocols. *Journal of Computer Security* 26, 1 (2018), 71–120. https://doi.org/10.3233/JCS-16814

[26] M. Marlinspike T. Perrin. 2016. *The Signal Protocol*. https://signal.org/ (Accessed May 29th, 2020).

[27] T. Y. C. Woo and S. S. Lam. 1993. A semantic model for authentication protocols. In *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*. 178–194.