Verification of Adaptable Pairwise Key Establishment Scheme using Scyther

Robert J. Dallara, Robert J. Lewis

***Abstract*—The Internet of things is increasingly allowing small embedded devices to monitor different aspects of the real-world in an increasingly coordinated fashion. Although this trend represents limitless opportunities for the future, such embedded devices might contain security vulnerabilities that could potentially compromise whole embedded device networks. The Adaptable Pairwise Key Establishment Schemes (APKES) is a recently proposed key establishment scheme which allows nodes in embedded systems to exchange pairwise keys with one another in order to ensure communication security. It solves many of the problems of other pairwise key schemes and has the flexibility of working with multiple pluggable schemes depending on what network it is deployed on. In this paper, we present (to the best of our knowledge) the first known formal verification of the APKES protocol itself. Although we only focus on certain properties of the protocol at a very high level, we did not find any attacks on the protocol suggesting its suitability for use in real-world embedded systems.**

***Index Terms*—formal verification, network security protocol, model checking**

I. Introduction

THE Internet of things represents enormous opportunity for the future of computing. It will allow ever more devices to be connected to the Internet, from smart sensors in homes to small devices which monitor industrial plants in real time [7]. In fact, the number of devices connected to the Internet of things is expected to grow to 26 billion by 2020, a 30-fold increase from 2009 [5]. As with any device connected to a network, embedded devices need a reliable way to connect to the outside world and with one another, and this is done through the IEEE 802.15.4 standard [6]. For even greater efficiency, many embedded nodes are increasingly being given the ability to directly communicate via IPv6 themselves (without having to rely on a "central node" to communicate with the outside Internet).

In order to reliably connect such embedded devices to the outside Internet in such a decentralized way, the 6LoWPAN ("IPv6 over Low-power Wireless Personal Area Networks") protocol stack [8] was created in order to allow embedded devices on low-power, 802.15.4-based wireless sensor networks to connect directly to the Internet via IPv6. It does so via an adoption layer between the IPv6 and link-layers in the stack [9, 10]. This layer both fragments and compresses incoming IPv6 packets in order to ensure that the embedded nodes on an 802.15.4 network can process them using less memory and with less energy consumption (which are special constraints especially unique to embedded devices).

Although the Internet of things represents an enormous opportunity, embedded devices and the protocols they use are vulnerable to potential attacks due to unforeseen flaws in designs and physical tampering. For example, if an embedded device containing a shared secret with other nodes is physically compromised, its information could be extracted, leaking the shared secret to the attacker. He/she could then use this secret to cause unintended if not malicious behavior among the nodes in the embedded device network [10]. Attacks from outside the network could also cause undesirable behaviors of the protocols that embedded devices use for communication, also potentially yielding secret information or perhaps draining node batteries as to cause a new form of denial-of-service (DOS) attack. Although a variety of pairwise and public key schemes have been proposed to deal with these security risks, most are not suitable given the limited memory and energy available in embedded network nodes. These issues will be discussed in further detail in Section 2.1 of this paper.

In order to best resolve these issues, Krentz *et al.* [10] have proposed a scheme known as APKES (Adaptable Pairwise Key Establishment Scheme), which is a protocol for establishing pairwise keys between two different embedded devices without using public-key cryptography. APKES is merely the protocol used to exchange some shared secret between two different nodes in order for them to establish pairwise keys: the actual secret used can be formed using a variety of different plug-in schemes. This allows APKES to adapt to the security and resource needs of different 802.15.4 networks in the real world [10].

However, to the best of our knowledge, APKES has not been formally verified. In order to address this issue, we decided to use a model checker known as Scyther [1], designed to formally verify network protocols. The tool works by iteratively sets of possible traces of the specified protocol in order to find possible counterexamples to verification claims. Protocol are modelled using a high-level, declarative language (Secure Protocol Description Language, or SPDL)

The authors both contributed equally to this paper as part of the Verified Security course offered by Dr. Cythnia Sturton at the University of North Carolina at Chapel Hill (Fall 2014).

R. J. Dallara is a graduate student at the University of North Carolina at Chapel Hill (e-mail: dallara@cs.unc.edu).

R. J. Lewis is a graduate student at the University of North Carolina at Chapel Hill (e-mail: rjlewis@cs.unc.edu).

and results, including any counterexamples, may be viewed in a convenient graphical interface.

Our major contribution in this paper is a formal verification of the APKES protocol using Scyther. In order to do this, we studied the APKES protocol as described by Krentz *et al*. [10] and formally modeled it in the SPDL language used in Scyther. As part of this model, we specified properties of secrecy, message agreement, and synchronization described in more detail by Cremers and Mauw [3] as part of the SPDL specification. We then used the tool's "verify" option in order to actually verify these particular protocol properties assuming a Dolev-Yao attacker [11] threat model. Although we did not find any attacks on the protocol in our particular analysis, we did not look into replay attacks or protocol implementation failures in the process.

The remaining structure of this paper is summarized as follows. Section 2 of this paper will discuss in more detail the background regarding both APKES and Scyther. Section 3 will describe our formal APKES verification in more detail, including assumptions made, more details about the properties we verify, and how we used Scyther to actually perform the verification. Section 4 will then discuss our results, what they mean in practice, and the process of choosing Scyther as our tool. Section 5 will then discuss other related work, and Section 6 will conclude the paper.

II. BACKGROUND

A. *APKES Protocol*

The 6LoWPAN protocol stack [8, 10] is designed to allow embedded nodes in 802.15.4-based sensor networks to communicate with one another and even with hosts on the outside Internet using IPv6 (on an individual basis, without having to go through a "central" node in the network). In particular, it is designed to integrate IPv6 with 802.15.4 sublayers by fragmenting and compressing incoming IP packets to make them suitable for use in the embedded device network below. The entire protocol stack is shown below in Figure 1, which was obtained from page 2 of the paper written by Krentz *et al*. [10].

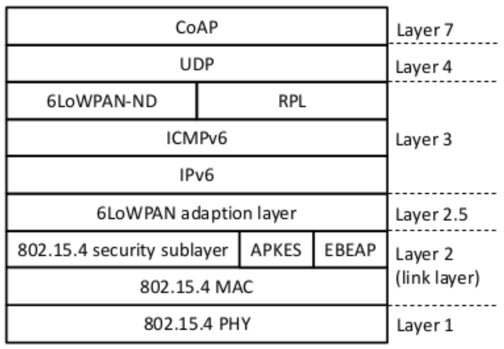


Figure 1: The 6LoWPAN protocol stack

The 6LoWPAN adaptation layer sits between the IPv6 and 802.15.4 layers, and is the layer most responsible for integrating 802.15.4-based wireless networks with IPv6. A security sublayer is also implemented between the IPv6 and link layers of the protocol stack. This layer is designed to verify the authenticity and "freshness" of different frames as they arrive at nodes in the 802.15.4 network [10]. In particular, it makes use of Message Integrity Codes (MICs) generated using CCM* [10] (a modified version of counter with CBC-MAC [12], which in turn uses AES [13]) [12] and frame counters to prevent replay attacks. This layer is also supposed to allow for establishment of keys among network nodes for additional security as well.

The simplest scheme that the 802.15.4 security sublayer could use for node key establishment is a network-wide key shared among all the nodes [10]. This is a very insecure solution due to the fact that networks of embedded devices usually run unattended in their particular setting, allowing an attacker to physically tamper with nodes in the network without being detected. Such an attack could allow the adversary to obtain the key on a single node, and hence be able to inject arbitrary frames into the network in arbitrary ways in the future. Although embedded device hardware could in theory make embedded nodes more secure [14], it can still be vulnerable to attack and is rather expensive to implement [15].

A more secure scheme for pairwise key establishment among nodes is the use of fully pairwise keys, but this consumes too much memory on 6LoWPAN networks with a large number of nodes. Since the number of pairs of nodes goes up at an O(n^2) rate as the number of nodes in a network of embedded nodes increases at a linear rate O(n), individual nodes in networks with hundreds of nodes might each have to store thousands of pairwise keys, taking up significant memory on small embedded nodes that already have very limited memory available.

As a result of this issue, the use of public key cryptography seems much more promising, as it only requires each node to have a single public and private key stored on it. However, this approach is too time and energy-consuming for embedded nodes to handle well. Even if key establishment happens only once (or very rarely), attackers could potentially take advantage of this drawback by repeatedly sending phony key establishment requests to embedded nodes in a 802.15.4 network, causing nodes to spend too much time on this task and even draining their batteries to effectively launch a denial-of-service (DOS) attack.

As shown in Figure 1, APKES sits in the same layer of the 6LoWPAN protocol stack as the 802.15.4 security sublayer. The job of APKES is to establish pairwise keys between two different nodes in order to ensure secure communication. APKES does this by means of a shared secret between the two nodes involved, and this shared secret can be generated by different pluggable schemes which APKES can work in conjunction with. For example, an upcoming version of the Contiki [16] operating system which Krentz is working on [17] makes use of the Localized Encryption and

Authentication Protocol (LEAP) scheme. This protocol uses a master key loaded into the two nodes in order to generate a shared secret between them. In order for this to work, a sender *u* first broadcasts an unauthenticated, unencrypted message containing some unique attribute ID_u of itself (*e.g.* its address) to all nodes, and then a neighboring node *v* sends back a packet containing the unique identifiers of itself and *u* (authenticated with its individual key K_v). Both *u* and *v* then calculate their pairwise key using the result of $F(K_v, ID_u)$, where *F* is a pseudorandom function family. The master key is then deleted to prevent it from being leaked in the event of a node compromise [10].

The APKES protocol solves the problems of other key schemes on the stack by avoiding public-key cryptography and by taking advantage of the fact that many inter-node communications on 802.15.4 wireless networks happen between nodes that are just one hop away [10]. In addition, no network-wide shared secrets (if used in a pluggable scheme, such as LEAP) are maintained across the two nodes during the entire duration of the protocol: they are deleted immediately after their use. The protocol itself is outlined in the figure below obtained from Section 4.1 of the paper by Krentz *et al*. [10].

$$\begin{aligned}
u &: \text{Generate } R_u \text{ randomly}\\
u \rightarrow * &: \texttt{HELLO}\ \langle R_u \rangle\\
v &: \text{Generate } R_v \text{ randomly and wait for } T_w \leq M_w\\
v &: K_{v,u} = \text{see Table 1}\\
v \rightarrow u &: \texttt{HELLOACK}\ \langle R_u, R_v \rangle_{K_{v,u}}\\
v &: K'_{v,u} = \text{AES}(K_{v,u}, R_u \| R_v)\\
u &: K_{u,v} = \text{see Table 1}\\
u &: K'_{u,v} = \text{AES}(K_{u,v}, R_u \| R_v)\\
u \rightarrow v &: \texttt{ACK}\ \langle\rangle_{K'_{u,v}}
\end{aligned}$$

Figure 2: the APKES three-way handshake

In particular, APKES makes use of a three way handshake between a sender node *u* and a receiver node *v*. After the symmetric shared secrets $K_{u,v}$ (the shared secret *u* has with *v*) and $K_{v,u}$ (the shared secret *v* has with *u*) are created using one of the supported pluggable schemes for APKES, *u* starts the protocol by generating a random nonce R_u and sending it (without any authentication or encryption) in a *HELLO* to all other nearby nodes in the network. A neighboring node *v* upon receiving this message then generates a random nonce R_v and waits for a random amount of time T_w (up to a maximum limit M_w) before sending a *HELLOACK* message back to *u*. This is done in order to prevent the original sender node *u* from being overrun with *HELLOACK* messages [10]. The *HELLOACK* message itself contains the two random nonces R_v and R_u authenticated with the shared secret $K_{v,,u}$. Since the nodes are often only one hop away, this shared secret can be used to both authenticate the *HELLOACK* and encrypt its payload. If the nodes were multiple hops away, this authentication scheme would be insecure as intermediate nodes could potentially decrypt the payload and leak data to the adversary [10].

As this part of the APKES protocol is taking place, *v* tries to generate its pairwise key $K^{'}_{v,u}$ using the Advanced Encryption Standard (AES) value of R_u || R_v using $K_{v,u}$ as its key. Upon receiving the *HELLOACK* from *v*, node *u* makes sure that the random nonce R_u in the incoming packet remains unchanged from before and that the packet's MIC generated from the packet contact earlier is intact as well. If all this goes well, *u* then generates its own pairwise key $K^{'}_{u,v}$ in the same way that *v* does earlier, except that it uses the $K_{u,v}$ key to encrypt R_u || R_v with AES. Finally *u* sends an *ACK* packet with an empty payload authenticated in the same was as before with this pairwise key $K^{'}_{u,v}$ back to node *v*. Assuming the packet arrives at *v* unchanged, *v* will have confidence that *u* received its data without any tampering in between. Node *u*'s confidence comes from the authentication of the *HELLOACK* packet and checking of nonce R_u earlier.

B. Scyther Model Checker

Scyther is a network protocol checker that allows a protocol designer or verifier to specify the protocol in a high-level, declarative language, SPDL [1, 2]. Protocols in SPDL are made up of roles in which events (*e.g.* send and receive messages) occur to accomplish the goals of the protocol and claims (*e.g.* secrecy of a key) about the truth of a statement at a given point in the execution of a role's events. It is important to note that a role is not a concrete actor, but rather a representation of a class of actors which send and receive similar messages. During verification, Scyther is able to perform unbounded verification/falsification via complete classification, the creation of a finite set of representative behaviors that include a given role [4].

Characterization is done by examining all the possible execution traces in which the total order of protocol events holds. Traces are grouped into patterns, which are directed acyclic graphs (DAGs) in which the partial order of events in a given pattern generalizes the total order of events represented by traces belonging to that pattern. Realizable patterns are those patterns in which traces of the actual protocol can be constructed, *i.e.* the total order of a protocol traces can be constructed from the partial order represented by an individual pattern. A process of pattern refinement manipulates patterns in a structured way in an attempt to construct realizable patterns. Verification proceeds by examining a pattern where a claim fails and attempting to refine it into a realizable pattern.

Many protocols can be characterized by a finite set of patterns, allowing Scyther to perform unbounded verification and falsification. When a claim does not hold, Scyther is able to use the characterizations to provide a counterexample. In situations where the number of patterns exceeds a predefined threshold, Scyther will perform a bounded verification for the number of patterns, stating that a claim holds or providing a counterexample. The net result is that Scyther is guaranteed to terminate and provide a counterexample in the case where a claim does not hold.

Scyther also provides multiple features to aid analysts in protocol verification. This includes an implicit Dolev-Yao style attacker, which handles the transmission of all messages as well as generate arbitrary messages. Additionally, while the attacker cannot break cryptography, it is able to generate messages from agents that it compromises and gain full knowledge of their secret keys. Lastly, the tool provides a public and secret key infrastructure, so a protocol analyst does not have to explicitly build one out for each protocol.

III. APKES Verification

Our verification effort comprised of a verification of APKES in the face of a Dolev-Yao style attacker. We do not attempt to verify a specific plugin scheme and instead use Scyther's built-in secret key infrastructure to simulate the initial shared secret used to generate the pair-wise keys. This constitutes an assumption that the APKES plugin schemes themselves are secure. We also did not attempt to model any replay attacks or the frame counters used to mitigate them. The verification effort examined the following claims (see [3] for formal claim definitions):

- Non-injective synchronization - All messages are sent in the correct order and with correct contents, even in the presence of an adversary over a single run of the protocol. An attacker replaying messages from a different context (*i.e.* performing a replay attack) may still be able to fool honest participants.
- Secrecy - The generated pairwise key is not known to the attacker after the protocol is executed.

Scyther was able to perform an unbounded verification and for each role (*u* and *v*) found only a single realizable trace pattern, meaning that there is only one way that the protocol (as modeled) can function. The verification of non-injective synchronization means the exchange of messages in APKES occurs as intended, even in the presence of an attacker. Finally, and most importantly, the generated pairwise key remains secret throughout the protocol.

IV. Evaluation

After creating our SPDL specification of the APKES protocol [18], we verified it in Scyther using the "Verify protocol" option under the "Verify" menu. In particular, our SPDL code assumes that the pairwise keys generated on both sides of the protocol communication are kept secret, and that non-injective synchronization and non-injective agreement [3] hold on both parties throughout the protocol. Upon verification, the SPDL tool found no attacks on our specified version of the protocol (and did a complete verification in the process), suggesting that there are no attacks on APKES as portrayed in Krentz *et al*. [10] involving the claims we looked at.

However, this does not mean APKES is completely free of security vulnerabilities. For example, packets in APKES make use of frame counters [10] in order to prevent replay attacks. These are used to prevent nodes from accepting packets that have already been used before in previous runs of the protocol. If an attacker were to try and re-use such a packet, it should be rejected by the protocol. However, flaws in the way frame counters are designed and implemented could still potentially lead to replay attacks.

Other attacks might potentially occur on systems that have implemented APKES incorrectly. For example, a node which does not always properly wait long enough [10] after receiving a *HELLO* before sending a *HELLOACK* might be able to overwhelm a sender node with *HELLOACK*s. An attacker might be able to take advantage of this if he/she were able to take control of a node that receives a *HELLO* from a sender at some point during the operation of the embedded network. An attacker might also be able to get around replay protection if the random nonce generation taking place at different is too predictable (and the attacker gleans knowledge of how it works, as might happen if the attacker is an insider to the network).

Another class of attacks is also possible if secret information used to generate shared secrets between two nodes in APKES pluggable schemes is leaked to an attacker. For example, if the master key used to generate a shared secret between two nodes in the LEAP pluggable scheme is leaked to an adversary, he/she could figure out the individual key of a node *v* which a newly deployed node *u* is communicating with at the time [10], assuming the unique identifiers of the two nodes involved are publicly known. Assuming the attacker knows how LEAP works and how the master key fits in with LEAP pairwise key generation, he/she could calculate the nodes' pairwise key (which will be the shared secret between the two nodes in APKES) and thus learn APKES' shared secret.

Finally, we discuss our choice of Scyther over other model checking tools. The first tool we considered using was CBMC [19], as we were originally planning to solely focus on verifying Contiki's upcoming implementation of APKES [17]. However, CBMC turned out to be inappropriate for our project due to complications setting it up to run on Contiki [16] as well as the interrupt-driven nature of such embedded operating systems. A similar tool we briefly considered using was the FIE tool [20], but it does not scale well too much larger embedded systems such as Contiki. We also considered using a tool known as ProVerif [21], but did not end up using it due to its complex, hard-to-read error reporting. We also considered using the Uclid tool [22], but chose against it since it seemed more suitable for verification of microprocessors than high-level Internet protocols such as APKES. We ultimately chose Scyther over all these other model checking tools because of its speed, strong suitability for verifying Internet-based protocols, and straightforward error reporting that makes use of visual diagrams to convey attacks to the user.

V. Related Work

Taha *et al*. [23] use the Scyther tool to verify the Privacy and Key Management (PKM) protocol used in broadband Internet connections taking place over long distances. They analyze both an older as well as newer version of this protocol and find a number of potential attacks, as well as come up with some possible defenses to the attacks they found. Similar to our project, they do this by specifying specific properties to verify and putting them into Scyther along with the actual protocol specification.

Davidson *et al*. implement a tool called FIE [20] for analysis of small firmware programs in embedded devices. It employs symbolic execution to analyze possible program execution paths as it looks for possible security violations. It employs state pruning to avoid analyzing equivalent program states multiple times as well as “memory smudging” to more effectively deal with long program loops by replacing loop counters with symbolic variables after a certain number of iterations. This tool is designed for small implementations of firmware, and not large embedded operating systems or high-level protocols used in embedded device networks.

Other more recent works also focus on improving the security of 6LoWPAN itself. For example, Hummen *et al.* [24] focus on the security of the 6LoWPAN packet fragmentation mechanism. They discuss possible attacks in which an attacker can prevent correct packet reassembly on a target node as well as two defense mechanisms against these attacks. Krentz et al. also propose the SCREWED mechanism for preventing certain security vulnerabilities in wireless sensor networks by trying to ensure channel reciprocity among network nodes [25]. They integrated this mechanism into the 6LoWPAN protocol stack of the Contiki operating system, and more specifically into the APKES protocol.

VI. Conclusion & Future Work

Our verification of APKES found no attacks involving the properties we specified, suggesting a high level of security and usability for the protocol scheme in real-world networks of embedded devices. Our analysis also indicated that Scyther, despite its limitations in analyzing certain aspects of protocols, is an excellent model checker for real-world internet protocols. Future work for this verification could include verifying different pluggable schemes (such as the LEAP protocol) in conjunction with APKES. This could be done through Scyther's support for feasibly analyzing “multi-protocol” attacks [1], in which protocols interact with one another and adversaries could use information gleaned in one protocol to attack another. In addition, an even deeper analysis of the Scyther protocol with a different model checker (such as the Tamarin model checker [26]) could be used to search for possible replay attacks in the APKES protocol as well as methods for mitigating them. Nonetheless, we present the first known formal verification focusing on the APKES protocol itself to the best of our knowledge.

References

[1] Cremers, Cas JF. "Unbounded verification, falsification, and characterization of security protocols by pattern refinement." In Proceedings of the 15th ACM conference on Computer and communications security, pp. 119-128. ACM, 2008.

[2] Cremers, Cas JF. "The Scyther Tool: Verification, falsification, and analysis of security protocols." In Computer Aided Verification, pp. 414-418. Springer Berlin Heidelberg, 2008.

[3] Cremers, Cas, and S. Mauw. "Security Properties." In Operational Semantics and Verification of Security Protocols, 37-65. Berlin: Springer, 2012.

[4] Doghmi, Shaddin F., Joshua D. Guttman, and F. Javier Thayer. "Skeletons, homomorphisms, and shapes: Characterizing protocol executions." Electronic Notes in Theoretical Computer Science 173 (2007): 85-102.

[5] Rivera, Janessa, and Rob Van Der Meulen. "Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020." Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. December 12, 2013. Accessed December 11, 2014.

[6] "IEEE Standard 802.15.4d." 2009.

[7] Kim, Eunsook, and Dominik Kaspar. "Design and Application Spaces for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)." (2012).

[8] Mulligan, Geoff. "The 6LoWPAN architecture." In *Proceedings of the 4th workshop on Embedded networked sensors*, pp. 78-82. ACM, 2007.

[9] Hui, Jonathan, and Pascal Thubert. "Compression format for IPv6 datagrams over IEEE 802.15. 4-based networks." (2011).

[10] Krentz, Konrad-Felix, Hosnieh Rafiee, and Christoph Meinel. "6LoWPAN security: adding compromise resilience to the 802.15. 4 security sublayer." In *Proceedings of the International Workshop on Adaptive Security*, p. 1. ACM, 2013.

[11] Dolev, Danny, and Andrew C. Yao. "On the security of public key protocols."*Information Theory, IEEE Transactions on* 29, no. 2 (1983): 198-208.

[12] Whiting, Doug, Niels Ferguson, and Russell Housley. "Counter with cbc-mac (ccm)." (2003).

[13] Miller, Frederic, Agnes Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[14] Becher, Alexander, Zinaida Benenson, and Maximillian Dornseif. *Tampering with motes: Real-world physical attacks on wireless sensor networks*. Springer Berlin Heidelberg, 2006.

[15] Anderson, Ross, and Markus Kuhn. "Tamper resistance-a cautionary note." In *Proceedings of the second Usenix workshop on electronic commerce*, vol. 2, pp. 1-11. 1996.

[16] "Contiki: The Open Source OS for the Internet of Things." Contiki: The Open Source Operating System for the Internet of Things. Accessed December 11, 2014.

[17] Krentz, Konrad-Felix. "Kkrentz/contiki." GitHub. Accessed December 11, 2014. https://github.com/kkrentz/contiki/tree/master/core/net/llsec/coresec.

[18] Lewis, Rob, and Rob Dallara. "Superdude264/Contiki_CoreSec_Verification." GitHub. 2014. Accessed December 11, 2014. https://github.com/superdude264/Contiki_CoreSec_Verification.

[19] "CBMC Homepage." The CBMC Homepage. Accessed December 12, 2014.

[20] Davidson, Drew, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution." In *USENIX Security*, pp. 463-478. 2013.

[21] Blanchet, Bruno. "An efficient cryptographic protocol verifier based on Prolog rules." In Computer Security Foundations Workshop, IEEE, pp. 0082-0082. IEEE Computer Society, 2001.

[22] Lahiri, Shuvendu K., and Sanjit A. Seshia. "The UCLID decision procedure." In*Computer Aided Verification*, pp. 475-478. Springer Berlin Heidelberg, 2004.

[23] Taha, Ahmed M., Amr T. Abdel-Hamid, and Sofiène Tahar. "Formal verification of IEEE 802.16 security sublayer using Scyther tool." In *Network and Service Security, 2009. N2S'09. International Conference on*, pp. 1-5. IEEE, 2009.

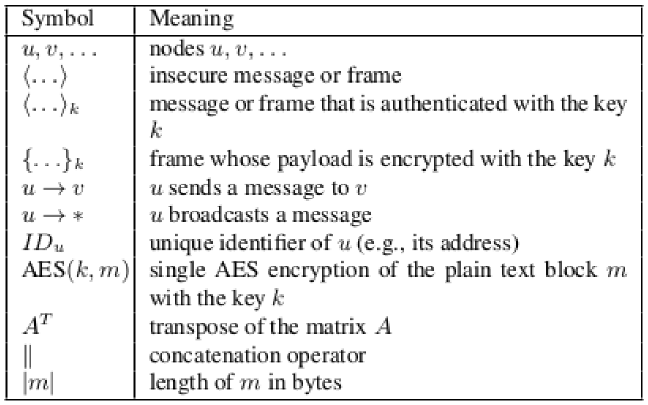
[24] Hummen, René, Jens Hiller, Hanno Wirtz, Martin Henze, Hossein Shafagh, and Klaus Wehrle. "6LoWPAN fragmentation attacks and mitigation mechanisms." In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pp. 55-66. ACM, 2013.

[25] Krentz, Konrad-Felix, and Gerhard Wunder. "6LoWPAN Security: Avoiding Hidden Wormholes using Channel Reciprocity." In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pp. 13-22. ACM, 2014.

[26] Meier, Simon, and Benedikt Schmidt. "Tamarin Prover." ETH - Institute of Information Security - Software. Accessed December 11, 2014.

APPENDIX

A. Notations



Symbol	Meaning
$u, v, \ldots$	nodes $u, v, \ldots$
$\langle \ldots \rangle$	insecure message or frame
$\langle \ldots \rangle_k$	message or frame that is authenticated with the key k
$\{\ldots\}_k$	frame whose payload is encrypted with the key k
$u \rightarrow v$	u sends a message to v
$u \rightarrow *$	u broadcasts a message
ID_u	unique identifier of u (e.g., its address)
$\mathrm{AES}(k, m)$	single AES encryption of the plain text block m with the key k
A^T	transpose of the matrix A
$\|\|$	concatenation operator
$\|m\|$	length of m in bytes

Table 1: From Krentz et al.