Duck Attack on Accountable Distributed Systems
Amrit Kumar, Cédric Lauradoux, Pascal Lafourcade

To cite this version:
Amrit Kumar, Cédric Lauradoux, Pascal Lafourcade. Duck Attack on Accountable Distributed Systems. 14th International Conference on Mobile and Ubiquitous Systems: Networks and Services MobiQuitous 2017, Nov 2017, Melbourne, Australia. pp.1-8. hal-01589196

HAL Id: hal-01589196
https://inria.hal.science/hal-01589196
Submitted on 12 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Duck Attack on Accountable Distributed Systems

Amrit Kumar∗
National University of Singapore
Singapore
amrit.kumar@polytechnique.org

Cédric Lauradoux
Inria
France
cedric.lauradoux@inria.fr

Pascal Lafourcade
Université Clermont Auvergne
France
pascal.lafourcade@uca.fr

ABSTRACT
Accountability plays a key role in dependable distributed systems. It allows to detect, isolate and churn malicious/selfish nodes that deviate from a prescribed protocol. To achieve these properties, several accountable systems use at their core cryptographic primitives that produce non-repudiable evidence of inconsistent or incorrect behavior. In this paper, we show how colluding adversaries can exploit the use of cryptographic digests in accountability protocols to mount what we call a duck attack. In a duck attack, colluding adversaries exploit the use of cryptographic digests to alter the transmission of messages while still looking honest: selfish behaviors remain undetected. We first discover the duck attack while analyzing PAG — a custom cryptographic protocol for accountable systems presented at ICDCS 2016. We later discover that accountable distributed systems based on tamper-evident log are also vulnerable to the duck attack and apply it on AcTinG — a protocol presented at SRDS 2014. To defeat our attack, we modify the commonly used hash-based log structure to have high-order dependency level between the authenticators and the messages stored in the log.

KEYWORDS
ACM proceedings, B\LaTeX, text tagging

ACM Reference format:
DOI: 10.1145/nmnmnn.nmnnnn

1 INTRODUCTION
Distributed systems are often plagued by nodes exhibiting selfish and malicious behavior. Selfish (rational) nodes may deviate from the prescribed protocol as and when there is an incentive to do so. Malicious or byzantine nodes can deviate arbitrarily from the protocol without having any well defined incentive. The end result is that the quality of the service to honest nodes gets affected and very often the honest nodes do not get served at all.

To this end, several approaches [HKD07, LCM+08, BDHU09, GHK+10, MDQ14a] have been proposed in the past to force these “rebellious” nodes to be compliant (to the underlying protocol) and to make them accountable for their actions in the network. By making every node accountable, nodes do not have incentive to deviate as any such attempt will eventually be detected resulting in its churn out from the network.

While considering accountability, attacks resulting from collusion between nodes are certainly hard to thwart as evidenced in [ELW11, WA13]. This is because colluding nodes generally perform unobservable actions from the point of view of the protocol making their deviations difficult to deter. In fact, this is also why the number of accountable protocols capable of handling colluding nodes is rather limited. FlightPath [LCM+08] fights collusion using Tit-for-Tat incentives. LiFTinG [GHK+10] uses cross checking and statistical analysis. Finally, AcTinG [MDQ14a] and PAG [DMPQ16] rely on cryptography to ensure that nodes’ actions are bound, non-repudiable, tamper-evident and verifiable [YC04, YC05]. This last category of solutions is the focus of our work. We analyze the security provided by AcTinG [MDQ14a] and PAG [DMPQ16] and discover a new attack.

Contributions: The main contribution of this paper is the design of a new form of collusion attack: the duck attack. The duck attack is named after the duck test1 and it can be summarized as follows: if you provide me the cryptographic digest \( H(m) \) of a message \( m \), then it probably means that you know \( H(m) \) and the message \( m \). Note that, this is outrightly false because concluding that you know \( m \) from the fact that you know \( H(m) \) is incorrect. We find this logical flaw in PAG [DMPQ16] and AcTinG [MDQ14a]. In a duck attack, the colluding nodes execute the protocol without exchanging the messages. Instead, they exchange the message digests and later commit the messages, when it is more advantageous to do so. For instance, they can use data compression techniques to send batches of messages and therefore save bandwidth. The duck attack is very similar to terrorist fraud [ABK+11] and distance hijacking attacks [CRC12] against authentication and distance bounding protocols, where colluding attackers attempt to confuse an authentication server.

To fight the duck attack, we revisit the tamper-evident log structure of AcTinG, which is a secure log initially proposed in [Pet02] and later also employed in several other protocols such as in PeerRev [HKD07]. Our approach consists in modifying the secure log. In AcTinG, a secure log is maintained by each node and each entry (message sent by the node) of the secure log yields an authenticity. The authenticator attests the creation of the log entry and can be publicly verified to detect deviation from the protocol. In our countermeasure, the computation of the authenticator depends on all the previous messages stored in the secure log. Cheating with the duck attack is prevented because one of the colluding parties

∗Work done while at Université Grenoble Alpes and Inria, France

1https://en.wikipedia.org/wiki/Duck_test
cannot maintain the log only by herself: She needs to make a lot of communication with her accomplice.

Outline: The next section of this paper is dedicated to the presentation of the three protocols that we attack in this paper. In Section 2, we discuss the threat model and colluding adversaries. We first demonstrate the power of the duck attack on PAG [DMPQ16] (Section 3). In Section 4, we also illustrate the duck attack on AeTiNoG a protocol which inherits the secure log system of PeerReview [HKD07]. Our log system with extended dependency is presented in Section 5 to thwart the attack. Finally in Section 6, we conclude the paper and mention some future works.

2 THREAT MODEL

We consider a network composed of correct nodes, individual (non colluding) malicious nodes and malicious colluding nodes. Correct nodes respect the normal execution of the protocol. Individual malicious nodes tamper with the execution of the protocol. They can modify the messages they receive and forward the modified version to the rest of the network for instance. Malicious colluding nodes can behave exactly as individual malicious nodes to harm the system. The colluding nodes can also have a rational behavior (see [AAC+05]). They follow the protocol if it is not in their interest but they are willing to deviate from it if they can save resources like bandwidth or computation. In order to do so, they can communicate using off-the-record channels or use covert channels [ELW11] to exchange data beyond those prescribed by the underlying protocol.

We assume throughout the paper that nodes are set up with some long term asymmetric cryptographic keys. We assume that nodes do not share their long term private keys with anybody. This is a limit to which the colluding nodes can share information to cheat. This assumption is implicitly made in most papers like PeerReview [HKD07]. Without it, colluding nodes sharing secrets would be equivalent to a single adversary controlling two nodes. A similar assumption is made in the analysis of authentication protocol resistant to certain form of collusion [ABK+11, CRC12].

3 DUCK ATTACK ON PAG (AND MORE)

PAG [DMPQ16] has been designed to provide accountability to fight both individual malicious nodes and malicious colluding nodes in gossip-based message dissemination schemes, where nodes periodically exchange data chunks with randomly chosen nodes [KMG03, BMM+08]. In addition, it provides privacy guarantees for the messages provided by the nodes. As our duck attack does not take advantage of the privacy mechanism, privacy properties are not detailed here and left in Appendix. The appendix also provides attacks on the privacy guarantees of PAG.

The main goal of PAG is to ensure that nodes respect the obligation-to-forward and the obligation-to-receive which are defined as (see [DMPQ16]).

- **Obligation-to-receive**: At a given communication round, a node must receive the messages sent by its predecessors that it received.
- **Obligation-to-forward**: A node must forward the messages it received at a given communication round $R$ to all its successors during round $R + 1$.

We describe how PAG enforces these obligations.

3.1 PAG’s Description

3.1.1 Assumption and setup. Consider a simple network composed of five nodes: $A$, $B$, $C$, $D$ and $E$ (see Fig. 1). The point of interest in this network is $D$ which receives messages (in the form of updates) from the producers $A$, $B$ and $C$. In practice, $A$, $B$ and $C$ may just forward the messages received from a previous hop rather than creating them. The expected task of $D$ is to forward the messages to $E$. The nodes $A$, $B$ and $C$ are called predecessors of $D$, while the node $E$ is referred to as a successor of $D$.

![Figure 1: Gossip based message dissemination.](image)

PAG assumes secure channels between the nodes: all node-to-node communications are encrypted with the help of a public-key infrastructure. Messages are also digitally signed to ensure their authenticity. However, in order to ease the presentation, we drop all node-to-node encryptions and the accompanied signature.

The core primitive used in PAG is a keyed homomorphic hash function. The function allows a witness (an auditor) to perform the privacy preserving verification. Following the notation used in [DMPQ16], we denote this hash function by $H_p$, where $p$ is the key. The hash function is homomorphic in the sense that for any two messages $u$ and $v$, and keys $p$, $p_1$ and $p_2$, the following two properties hold:

\[
H_p(u) \cdot H_p(v) = H_p(u \cdot v),
\]

\[
H_{p_1}(H_{p_2}(u)) = H_{p_1 \cdot p_2}(u).
\]

In PAG, $H_p$ is instantiated by a variant of the RSA function with a prime exponent $p$ and a modulus $M$. Hence, for any message $u \in \{0, 1\}^*$, the digest of the hash function $H_p$ is given as:

\[
H_p(u) = u^p \mod M.
\]

The modulus is chosen once at the start of the protocol, while a different key (the exponent) is used for every predecessor $D$.

The keys are also updated at the start of each round.

3.1.2 Protocol Core. We continue with the example network of Fig. 1 with six nodes: $A$, $B$, $C$, $D$, and $E$, where $D$ is the point of interest. It has three predecessors $A$, $B$ and $C$ and a successor $E$. We augment this network with a witness $W$ for $D$.

The protocol runs in rounds and assumes that the network is synchronous. A schematic representation of the messages exchanged in a round of PAG is given in Fig. 2. Step 1, each predecessor node asks for a key. $D$ hence generates three random keys $p_1$, $p_2$, $p_3$ and sends $p_1 \cdot p_2 \cdot p_3$ to $A$, $p_2 \cdot p_3$ to $B$ and $p_3 \cdot p_1 \cdot p_2$ to $C$. Step 2,
the predecessor nodes send their respective messages \( u_1, u_2 \) and \( u_3 \) to \( D \). Step 3, each predecessor node computes the homomorphic hash of its message with the key received from \( D \) and sends to the witness the computed digest along with the received product of two keys. For instance, \( A \) sends \( H_{p_1}(u_1), p_2 \cdot p_3 \) to \( W \). Meanwhile, upon reception of the messages from its predecessors, \( D \) forwards them to its successor \( E \) along with the product of all the keys. \( E \) then computes \( H_{p_1 \cdot p_2 \cdot p_3}(u_1 \cdot u_2 \cdot u_3) \) and sends the digest to \( W \).

![Figure 2: A round of PAG with three predecessors (A, B, C) and one successor (E). W is a witness for D.](image)

Upon reception of all the messages from the predecessors and the successor, \( W \) computes: \( x = H_{p_2 \cdot p_3}(H_{p_1}(u_1)), y = H_{p_1 \cdot p_3}(H_{p_2}(u_2)), \) and \( z = H_{p_1 \cdot p_2}(H_{p_3}(u_3)) \). Values \( x, y, \) and \( z \) are computed using the messages received from \( A, B \) and \( C \) respectively. The witness then checks for equality between \( x \cdot y \cdot z \) and \( H_{p_1 \cdot p_2 \cdot p_3}(u_1 \cdot u_2 \cdot u_3) \) (the last digest was sent by \( E \)). If the messages were correctly forwarded, the homomorphic properties of the hash function ensure that the two terms are equal. Indeed,

\[
x \cdot y \cdot z = H_{p_2 \cdot p_3}(H_{p_1}(u_1)) \cdot H_{p_1 \cdot p_3}(H_{p_2}(u_2)) \cdot H_{p_1 \cdot p_2 \cdot p_3}(u_3)
\]

\[
= H_{p_2 \cdot p_3}(H_{p_1}(u_1 \cdot u_2)) \cdot H_{p_1 \cdot p_3}(H_{p_2}(u_2)) \cdot H_{p_1 \cdot p_2 \cdot p_3}(u_3)
\]

\[
= H_{p_2 \cdot p_3}(H_{p_1}(u_1 \cdot u_2 \cdot u_3))
\]

If the equality test fails, then \( W \) may conclude that \( D \) did not correctly forward the messages to \( E \).

We note that the actual protocol is slightly more complicated since it assumes several witnesses for a given node and that the node \( E \) does not directly communicate with the witness. In fact, the witnesses for \( D \) and \( E \) communicate with each other to crosscheck the messages sent by the monitored nodes. The protocol simplification is only meant to ease the understanding of the presented attacks. It does not limit in any manner the scope of the attacks on the actual protocol.

### 3.2 Key Recovery Attack

It is argued in [DMPQ16] that in order to ensure the privacy preserving auditing of the nodes, the witness should not learn the keys generated by a node. Moreover, it is claimed in [DMPQ16] that if every node has three predecessors, then the witness can not learn the keys. We first show that this claim is unfounded by illustrating an attack on our example network of three predecessors Fig. 2. The goal of this attack is to further simplify the PAG protocol so that the presentation of our duck attack becomes easier.

The witness \( W \) receives \( p_1 \cdot p_2 \) from \( C \) and \( p_1 \cdot p_3 \) from \( B \) (cf. Fig. 2). In order to recover the key \( p_1 \), it simply computes \( \gcd(p_1 \cdot p_2, p_1 \cdot p_3) \). Since, the keys are prime numbers and generated randomly, this computation yields \( p_1 \). Other keys \( p_2 \) and \( p_3 \) can be obtained likewise. It is evident that this attack trivially extends to any network with arbitrary number of predecessors.

As a result, all the keys can eventually be recovered by the witness. Hence, the keyed hash function now reduces to an unkeyed hash function. Apart from being an attack, this result allows us to further simplify the protocol by assuming that instead of generating one key per predecessor, nodes create a unique key \( p \) per round. The key is public to any node participating in the network (valid under the assumption that nodes collude). This does not change the verification procedure at the witness node. In the rest of this section, we work with only two predecessors to simplify the presentation.

### 3.3 Duck Attack with Colluding Nodes

We now present the duck against PAG, where, a unique key known to the witness node is used. To this end, we assume the message transmission scenario as depicted in Fig. 3, where two predecessor nodes \( A \) and \( B \) send messages \( u \) and \( v \) respectively to \( C \). The node \( W \) is the witness for \( C \). We assume that nodes \( C \) and \( D \) collude. Hence, to save bandwidth \( C \) sends to \( D \) the digest \( H_p(v) \) instead of the message \( v \). Since, \( D \) colludes with \( C \), it does not raise alarm and sends \( H_p(u \cdot v) \) to the witness \( W \). The witness can not detect the discrepancy and declares the verification as successful. Hence, PAG does not ensure the obligation-to-forward guarantee.

![Figure 3: Duck attack on a round of PAG. C instead of forwarding the message \( v \) to \( D \), forwards \( H_p(v) \). In red, we show the message sent as an adversary.](image)
same message \( u \) to \( C \). The latter may inform \( B \) before the transmission of the actual message that it is about to receive or (has received) the same message from another of its predecessors. Hence, to save bandwidth \( B \) does not send the message to \( C \). However, it sends \( H_p(u) \) to \( W \) to convince the witness that it in fact went through with the transmission. \( C \) colludes with \( B \) and forwards two copies of \( u \) to \( D \) and the rest of the protocol round runs as any other. The protocol does not respect the obligation-to-forward.

**Figure 4:** Collusion attack on a round of PAG. Dashed arrow and the symbol \( \times \) is to denote that this transmission never occurred.

Clearly, with the existing mechanism in PAG, it is not possible for witnesses to detect such behaviors. The fundamental reason why this selfish behavior goes undetected in PAG is that the witness has no way to learn that the digest sent by \( D \) on the message \( u \) did originate from \( B \).

4 DUCK ATTACK ON ACTING

AcTinG [MDQ14a] is a protocol presented at SRDS 2014 and inspired by PeerReview [HKD07] and FullReview [DMAQ14]. These protocols rely on maintaining a secure data structure on each node which implements a tamper evident log. The log is used to record why this selfish behavior goes undetected in PAG is that the witness cannot predict when the audits take place in AcTinG.

AcTinG has several layers, we first describe how two nodes interact when a message is exchanged and how the duck attack applies to this exchange. We then explain how the colluding adversaries can predict when the audits take place in AcTinG.

4.1 Secure Log in AcTinG

Each node in AcTinG maintains a secure log. Let us assume that at time \( i \), node \( A \) has last entry \( T_i \) which contains an authenticator (later used for audit) \( T_i \) and a message \( m_i \) sent by it. Then, the next entry \( T_{i+1} \) is defined by:

\[
T_{i+1} = H(T_i || i+1 || m_i+1),
\]

with \( H \) a cryptographic hash function and \( T_0 \) a fixed value. Note that the authenticator chains all previous messages sent and generates a single digest. Checking the validity of an authenticator means checking that all previous messages have been correctly logged and that the log has not been tampered with.

We have simplified how AcTinG stores messages in the log. In fact, AcTinG uses a special format and structure for the message which is not relevant to our attack. Fig. 5 describes how the log entries are created during a communication between two nodes \( A \) and \( B \). In addition to the message \( m \), node \( A \) sends:

\[
\alpha_{i+1}^A = \sigma(i+1||T_{i+1}^A),
\]

with \( \sigma \) a digital signature algorithm. The value \( \alpha_{i+1}^A \) is used by \( B \) during the audit to check the correctness of \( A \).

**Figure 5:** Log entries created by AcTinG during the transmission of \( m \) by \( A \) to \( B \).

4.2 Luring AcTinG

The duck attack against AcTinG is based on the observation that node \( B \) does not need to know \( m \) to compute \( T_{j+1}^B \) but only \( H(m) \). Node \( A \) just needs to send \( H(m) \) to \( B \). Then, \( B \) can create a blank entry \( T_{j+1}^B \) which does not contain the message but has a valid authenticator \( T_{j+1}^B \) that can be used to compute \( T_{j+2}^B \). Node \( B \) can continue to create new entries in its log despite the absence of \( m \). If a node performs an audit of \( B \), then it will discover the blank entries. Therefore, it is important that before an audit, \( A \) sends \( m \) to \( B \). The attack is schematically presented in Fig. 6.

**Figure 6:** Duck attack for \( A \) and \( B \).

At the first sight, the duck attack may appear to be pointless: executing the attack strategy is more costly than following the protocol. In the duck attack, node \( A \) sends \( H(m) \) and then must late...
commit \(m\) to \(B\). The communication of \(H(m)\) is an extra overhead compared to the normal execution of the protocol. The duck attack is not interesting if the colluding nodes execute it for a single message. The attack is interesting if they use it for several messages. Let us assume that node \(B\) has created two blank entries with the help of \(A\). Each entry is associated with message \(m\) and \(m'\). Node \(A\) needs to commit \(m\) and \(m'\) before the audit of \(B\). Node \(A\) can use data compression to send \(m\) and \(m'\) with gain bounded by information theory argument and reduce the cost of communicating with \(B\). With many blank entries, the colluding nodes can even expect greater gain.

4.3 Predicting AcTinG’s Audit

It is now crucial for the nodes who want to mount a duck attack to know when the audit will be performed. When a node \(A\) is associated to a node \(B\), node \(A\) decides if it performs a full audit of \(B\) or not. The full audit of \(B\) implies to check:

- the secure log of node \(B\);
- the secure logs of all the nodes which have communicated with \(B\);
- \(B\) has forwarded all the messages it was supposed to;
- \(B\) has performed all the necessary audits.

The decision to run this audit is taken using a non-deterministic process:

- Node \(A\) chooses a signature \(a_i\) and computes:
  \[ r = a_i \mod 100 \]
- Node \(A\) computes:
  \[ r = H(PK_A \| PK_B \| i) \mod 100, \]
  with \(PK_A\) and \(PK_B\) the respective public key of \(A\) and \(B\) and \(i\) the current value of the counter.

If the value \(r\) is greater than a certain threshold then \(A\) audits \(B\). The paper suggests a threshold of 30%.

The first solution favors selfish nodes. \(A\) chooses \(a_i\) such that no audit takes place. The second solution is deterministic and predictable. Nodes \(A\) and \(B\) can compute in advance all the values \(H(PK_A \| PK_B \| i)\) for all the nodes to determine when they are going to be audited. Therefore, they can determine when they can execute the duck attack without any risk of being caught. The conjunction of the duck attack and the fact that the audit of AcTinG is predictable makes the risk analysis presented AcTinG [MDQ14a] too optimistic about the probability to detect a fraud.

5 COUNTERMEASURES

The authors of this paper do not believe it is possible to fix PAG to achieve both security and privacy. Additional security and privacy issues of PAG are presented in Appendix to support our opinion. In this section, we focus on fixing AcTinG. We propose two modifications of AcTinG to defeat the duck attacks: the usage of verifiable pseudo-random function and a modified secure log data structure.

In AcTinG, the nodes need to use randomness to decide when the audit are made and to choose the nodes they communicate with. In the current description of AcTinG, the random number generator is either deterministic or not accountable. It emphasizes the difficulty to achieve accountability when the nodes have a non-deterministic behavior. This problem was first tackled by Backes et al. [BDHU09] to extend PeerReview to non-deterministic nodes’ behavior. The random number used during the protocol must be unpredictable and verifiable. This property was achieved in CSAR [BDHU09] by using verifiable pseudo-random functions [MRV99]. It is the natural option to fix AcTinG. Guerraoui et al. [GKH+10] have proposed a different approach by measuring entropy, but the guarantees are not as strong as those provided by CSAR. Using a verifiable pseudo-random function ensures that adversaries cannot predict or influence the result of the random number generator without being detected.

AcTinG [MDQ14b], PeerReview [HKD07] and many others [BDHU09, DMAQ14] are based on the secure log structure proposed by Maniatis and Baker in [Pet02]. The security of their design is based on one-way hash function \(H\) (SHA-1 en 2002). Maniatis and Baker in fact propose a secure timeline log in [Pet02]. At time \(i\), the last entry in the log contains message \(m_i\) and is associated with authenticator \(T_i\). Then, the next entry is associated to \(T_{i+1}\) defined by:

\[ T_{i+1} = H(i+1|T_i||G(m_{i+1})). \]

\(T_0\) is a fixed value. The function \(G\) is also a one-way hash function. The security argument given to justify their design works as follows. Given \(T_{i+1} = H(i+1|T_i||G(m_{i+1})),\) it is not possible to produce a message \(\beta\) and an authenticator \(T'_{i} \neq T_i\) such that \(T_{i+1} = H(i+1|T'_{i}||G(\beta))\) since \(H\) is second pre-image resistant.

The authors of [Pet02] justified the use of \(G\) in their construction by the fact that the \(m_i\) can be very large (complete Merkle’s tree or any authenticated data structure). It is possible without loss of security to get rid of the function \(G\) as we assume now that \(H\) is a collision-resistant hash function (SHA-3 [Dwo15]). In our scheme, we compute the authenticator as follows:

\[ T_{i+1} = H(i+1|T_i||m_{i+1}|m_i|\cdots|m_1)). \]

The computation of \(T_{i+1}\) depends on all the previous messages stored in the log. For a given \(T_{i+1} = H(i+1|T_i||m_{i+1}|m_{i-1}|\cdots|m_1)),\) it is intractable for an adversary to find \(T'_i \neq T_i\) and/or \(\alpha \neq m_i\) such that:

\[ T_{i+1} = H(i+1|T'_i||m_{i+1}|\alpha|\cdots|m_1)), \]

because \(H\) is collision resistant. Therefore our modification is secure.

Let us consider two nodes \(A\) and \(B\) which exchange messages and maintain a log of their communication using our modification. We also assume that each time a node \(B\) receives a message \(m_i\) from \(A\), it sends to \(A\) an acknowledgement \(T_i^B||\sigma(T_i^B)\) with \(\sigma(T_i^B)\) a cryptographic signature. It appears that the duck attack still applies. The only difference with the duck attack previously described in this paper is that node \(B\) needs to provide to \(A\) the current authenticator of its log (see Fig. 7). At the end, \(B\) sends its signature to \(A\) to end the communication. The use of a randomized signature does not make the attack more difficult because \(A\) can recompute everything for \(B\) (except the final signature).

Our approach is not to detect or prevent directly the duck attack but to make it more costly than any benefit that two colluding parties can expect. Let consider a case in which node \(B\) colludes with node \(A\) and exchanges messages with an honest node \(C\). This situation is represented in Fig. 8. \(A\) and \(B\) first execute the duck
We have applied the duck attack on a custom cryptographic system with $\Delta$ is the use of cryptographic hash function as a core of more complex primitives. It allows attackers to exchange digests instead of the real messages and therefore be selfish. After discovering our attacks on PAG, we have been unable to propose corrections that fix all the flaws. It seems difficult to do so for PAG. We believe that designing a protocol that guarantees the accountability property and at the same time respects the privacy of the nodes might be very challenging. Designs based on a secure log can however be fixed by increasing the dependency order of the authenticators’ computation and by using verifiable pseudo-random function.

A key issue to go forward in the design of accountable distributed protocols is the definition of the selfish adversaries. Recently in [KTV10] Kuesters et al. presented a formal definition of accountability and some possible links with verifiability properties. It might be interesting to see how these definitions are sensitive to the duck attack and how they match selfish adversaries. In other words, we can investigate whether the two protocols studied in this paper satisfy their definitions or if the duck attack allows us to prove the contrary.

**ACKNOWLEDGMENT**

The work was partly supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025), the project-team SCCyPhy and the Digital Trust Chair from the University of Auvergne Foundation.

**REFERENCES**


---

**Figure 7:** Duck attack on the modified secure log structure.
A MORE ATTACKS ON PAG

During our analysis of PAG, we discover several other weakness in PAG. Some of these weaknesses can be fixed but at the cost of some important properties for PAG (like privacy for instance). We discover that the homomorphic hash function used by the authors is not collision free.
A.1 Hash Collisions

PAG is also insecure because the cryptographic primitive used to instantiate the function $H_p$ of the previous section has collisions. A trivial collision exists for two messages $u$ and $v$ such that $v = u + k \times M$ for any $k \in \mathbb{N}^*$:

$$H_p(v) = v^p \mod M,$$
$$= (u + k \times M)^p \mod M,$$
$$= u^p \mod M.$$

$$H_p(u).$$

We note that the RSA function defined using a modulus $M$ is a permutation on the set $[0, M - 1]$. Hence, a hash function defined using it should not yield collisions when the message space is restricted to $[0, M - 1]$. However, PAG assumes that any message $u$ exchanged between the nodes is larger than $M$. This assumption clearly leads to collisions.

A.2 From Collisions to Downgrade Attack

We consider the network depicted in Fig. 10. For this network, we also consider the following situation during a round: Node $A$ sends $u' = u + M$ to $C$ and node $B$ sends an arbitrary message $v'$. Node $C$ can substitute $u'$ by $u$ without being detected. Indeed, $C$ forwards $u$ and $v'$ to $D$. Then $D$ sends $H_p(u \cdot v')$ to $W$. We recall our assumption that a unique $p$ used throughout and is known to all the participants of the network.

\[ H_p(u') \]
\[ H_p(v') \]
\[ W \]
\[ H_p(u \cdot v') \]
\[ A \]
\[ B \]
\[ C \]
\[ D \]

Figure 10: Downgrade attack on a round of PAG. In red, we show the message sent as an adversary.

The witness $W$ upon receiving $H_p(u')$ and $H_p(v')$ from $A$ and $B$ respectively verifies that:

$$H_p(u') \cdot H_p(v') = H_p(u \cdot v') = H_p(u) \cdot H(v').$$

$W$ therefore concludes that $C$ has indeed respected the obligation-to-receive and the obligation-to-forward. In reality though, $C$ has not respected the obligation-to-forward by substituting $u'$ by $u$. The node $C$ has successfully downgraded $u'$ to $u$ by exploiting the collision.

A.3 Breaking Unlinkability

In this section, we present two attacks to break the unlinkability guarantee of PAG. Our first attack that we refer to as short-term linkability exploits the fact that PAG employs a small RSA modulus.

Our second attack that we refer to as long-term linkability exploits our earlier observation that keys of the hash function can be learned by the witness, hence making the function behave deterministically.

A.3.1 Short-Term Linkability. Contrary to the claim that PAG is preserving privacy, the witness can in fact invert the hash function and learn the message corresponding to a received digest. This allows the witness to link the message to the node that sent the digest and hence break the claimed unlinkability property.

Inversion of digests is possible since the hash function $H_p$ is instantiated with a modulus $M$ of size 512 bits. It is argued in [DMPQ16] that a 512-bit modulus suffices when the RSA function is used as a hash function. However, for a small modulus $M$ of 512 bits, it is possible to obtain the prime factorization of $M$ and compute $\varphi(M)$ where, $\varphi$ is the Euler’s totient function. Knowledge of $\varphi(M)$ allows to invert any digest $H_p(u)$ and obtain $u$ as:

$$H_p(u)^{p^{-1}} \mod \varphi(M) \mod M = u.$$

It is to note that an RSA modulus of 512 bits named RSA-155 was successfully factored by Herman te Riele et al. [CDL*00] as early as in 2000. The most recent RSA modulus factored is of 768 bits. The feat was achieved by Kleinjung et al. in 2010 [KAF+10]. In 2015, NIST published a standard [BR15] which recommends using a modulus of 2048 bits. These references clearly suggest that the choice of modulus size in PAG is inappropriate.

The linkability attack based on inverting the hash function holds as long as the modulus size is small enough. Choosing an appropriately large modulus prevents inversion. We hence refer to this attack as a short-term linkability attack.

A.3.2 Long-Term Linkability. Choosing a sufficiently large modulus however does not prevent other linkability attacks that the witness may mount. In fact, since the key of the hash function is known, the function now becomes deterministic. As a result, the witness can detect if a message is replayed by a node. It is also possible to relate two nodes if they send the same message. We refer to these attacks as long-term linkability attacks since they exist even when the modulus size is sufficiently large.

In order to understand the ensuing implications, let us consider a content sharing PAG network. If two different nodes send the same digest to a witness, then it may learn the interest graph between nodes sharing similar interests, thus possibly inferring private information about them.

\footnote{RSA-155 was a part of the RSA Factoring Challenge put forward by RSA Laboratories.}