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 selfish and colluding nodes can exploit the use of cryptographic digests in accountability protocols to mount what we call a *duck attack*. In a duck attack, selfish and colluding nodes exploit the use of cryptographic digests to alter the transmission of messages while masquerading as honest entities. The end result is that their selfish behavior remains undetected. This undermines the security guarantees of the accountability protocols.

We first discover the duck attack while analyzing PAG — a custom cryptographic protocol to build accountable systems presented at ICDCS 2016. We later discover that accountable distributed systems based on a secure log (essentially a hash-based data structure) 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 underlying secure log to have high-order dependency on the messages stored in it.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; *Domain-specific security and privacy architectures*;

KEYWORDS

Accountability, Duck attack, Secure log, Public verifiability

ACM Reference Format:
Amrit Kumar, Cédric Lauradoux, and Pascal Lafourcade. 2017. Duck Attack on Accountable Distributed Systems. In *the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3144457.3144480

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. For instance, in a live streaming system, selfish nodes would download the video stream faster while saving upload bandwidth [25].

To this end, several approaches [3, 15, 16, 20, 23] 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, selfish nodes do not have incentive to deviate as any such attempt will eventually be detected resulting in their churn out from the network.

While considering accountability, attacks resulting from collusion between nodes are certainly hard to thwart as previously evidenced in [12, 28]. This is because colluding nodes generally perform unobservable actions from the point of view of the protocol making their deviations difficult to detect. In fact, this is also why the number of accountable protocols capable of handling colluding nodes is rather limited. FlightPath [20] fights collusion using Tit-for-Tat incentives. LiFTinG [15] uses cross checking and statistical analysis. Finally, AcTinG [23] and PAG [8] rely on cryptography to ensure that nodes' actions are bound, non-repudiable, tamper-evident and verifiable [29, 30]. This last category of solutions is the focus of our work. We analyze the security provided by two recent accountability proposals that handle selfish and colluding nodes: PAG [8] and AcTinG [23].

PAG [8] is a privacy-preserving accountability protocol for gossip-based message dissemination, where, each node in the network is expected to forward a message (received from a previous hop) to a set of other nodes. The goal is to detect nodes that receive a message but do not forward it. It relies on a custom cryptographic protocol (details are given in Section 3.1). AcTinG [23] is a more general purpose protocol to detect nodes that do not comply with an underlying protocol. It requires nodes to log each message sent/received pertaining to the underlying protocol in a secure log that cannot be tampered with. The log is periodically audited by other nodes to detect any selfish/malicious behavior (details are given in Section 4).

In this work, we report a new kind of logical attack that we refer to as the *duck attack*. The attack can be mount on both AcTinG and PAG. Duck attack is named after the duck test[1] and it can be summarized as the following expression of abductive reasoning:

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

MobiQuitous 2017, November 7–10, 2017, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-5368-7/17/11…$15.00
https://doi.org/10.1145/3144457.3144480

[1] Duck test is a humorous term and has the following expression: "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck". The test implies that a person can identify an unknown subject by observing that subject's habitual characteristics. It is sometimes used to counter abstruse, or even valid, arguments that something is not what it appears to be. This definition is taken from https://en.wikipedia.org/wiki/Duck_test

> *If you provide me the cryptographic digest $H(m)$ of a message m, then it probably means that you know the message m.*

Note that this reasoning is incorrect because if you give me $H(m)$, the best that I should be able to say is that you know $H(m)$. I cannot conclude anything about your knowledge on m. To see this, consider the example of an authentication service that stores a user's password (m) in the form of a digest ($H(m)$). If the authentication service gives the password digest to a third party, say a password auditing service, it does not mean that the authentication service knows the user's password (assuming the password is long enough and well-chosen). Digest $H(m)$ only serves as a "commitment" to the message m. An entity holding the "commitment" may not necessarily know the underlying message.

We found this logical flaw in both the accountability protocols that we study in this work: PAG [8] and AcTinG [23]. Roughly speaking, the duck attack exists on PAG because nodes exchange $H(m)$ to prove that they have received a message m. As for AcTinG, the existence of duck attack is not so apparent. In fact, the attack exists due to the underlying secure-log, where a node logs $H(m)$ instead of logging m for every message m received or sent by it.

Duck attack in PAG and AcTinG manifests itself in the following manner: Two colluding nodes execute the protocol without exchanging the messages. Instead, they exchange the message digests and later transmit 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 [2] and distance hijacking attacks [7] against authentication and distance bounding protocols, where colluding attackers attempt to confuse an authentication server.

We also propose a countermeasure for AcTinG. To this end, we revisit the *tamper-evident log* data structure of AcTinG, which is a secure-log initially proposed in [26] and later also employed in several other protocols such as in PeerReview [16]. 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 *authenticator*. 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 in the extreme case on all the previous messages* stored in the secure log. Cheating through the duck attack is prevented because one of the colluding parties cannot maintain the log only by herself: the other accomplice cannot postpone anymore the sending of the messages and it will eventually cost a lot of communication.

While studying duck attack on PAG, we further discover other design flaws that lead to several other attacks on its security and privacy guarantees. This is an orthogonal contribution of our paper. For reasons explained in Section 5, we do not attempt to fix PAG.

Contributions: In summary, we make the following contributions in this paper:

(1) We devise a new form of collusion attack on accountable systems: the *duck attack*. Duck attacks requires selfish and colluding nodes – a standard adversary model supported by several systems such as PAG and AcTinG.
(2) We demonstrate how the duck attack can be mount on two state-of-the-art accountability protocols namely, PAG and AcTinG and the impact that they exert on the security guarantees of the respective systems.
(3) We provide a countermeasure for AcTinG and study the ensuing security-performance trade-off.
(4) An orthogonal contribution of this work is that we found several other attacks on the security and privacy guarantees of PAG. These attacks render PAG insecure.

Outline: In Section 2, we discuss the threat model and colluding adversaries. We first demonstrate the power of the duck attack on PAG (Section 3). In Section 4, we also illustrate the duck attack on AcTinG, a protocol that inherits the secure-log system of PeerReview [16]. Our log system with extended dependency is presented in Section 5 to thwart the attack. In Section 6, we conclude the paper and mention some future works. We leave in Appendix A the presentation of other attacks on PAG.

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 [1]). 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 [12] 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 [16]. 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 [2, 7].

3 DUCK ATTACK (AND MORE) ON PAG

PAG [8] 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 [5, 17]. 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 A. 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-receive* and the *obligation-to-forward* properties which are informally defined in [8] as follows:

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

In the next section, we describe how PAG enforces these obligations.

3.1 PAG's Description

Below, we first describe the assumptions, setup and the building block namely *keyed homomorphic hash function* and then present the PAG protocol.

3.1.1 Assumptions 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.

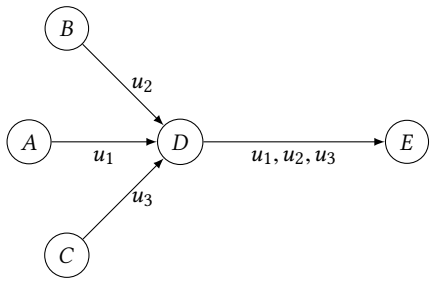

Figure 1: Gossip based message dissemination.

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 simplify the presentation, we ignore 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 [8], 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:

$$\begin{aligned} H_p(u) \cdot H_p(v) &= H_p(u \cdot v), \\ H_{p_1}(H_{p_2}(u)) &= H_{p_1 \cdot p_2}(u). \end{aligned}$$

The operation '$\cdot$' denotes product between the messages that come from a group.

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:

$$H_p(u) = u^p \bmod M.$$

The modulus is chosen once at the start of the protocol, while a different key (the exponent) is used for every predecessor node. 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 $p_1, p_2 \cdot p_3$ to A; $p_2, p_1 \cdot p_3$ to B and $p_3, p_1 \cdot p_2$ 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.

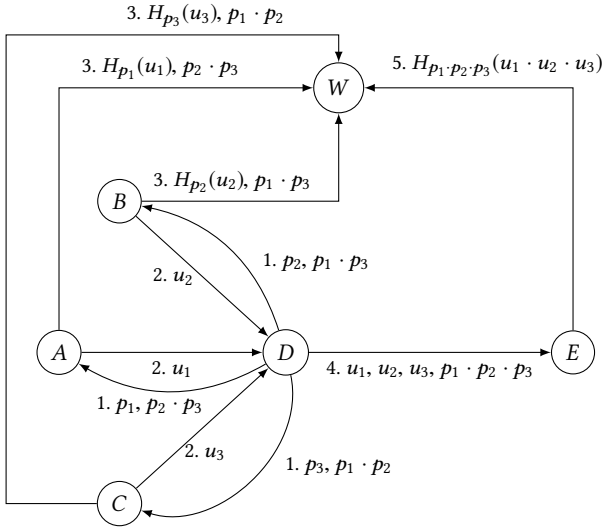

Figure 2: A round of PAG with three predecessors (A, B, C) and one successor (E). W is a witness for D.

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,

$$\begin{aligned} x \cdot y \cdot z &= H_{p_2 \cdot p_3}(H_{p_1}(u_1)) \cdot H_{p_1 \cdot p_3}(H_{p_2}(u_2)) \cdot H_{p_1 \cdot p_2}(H_{p_3}(u_3)) \\ &= H_{p_1 \cdot p_2 \cdot p_3}(u_1) \cdot H_{p_1 \cdot p_2 \cdot p_3}(u_2) \cdot H_{p_1 \cdot p_2 \cdot p_3}(u_3) \\ &= H_{p_1 \cdot p_2 \cdot p_3}(u_1 \cdot u_2 \cdot u_3) \end{aligned}$$

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 [8] 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 [8] 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 in 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 attack 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 and we color them in gray in Fig. 3. Hence, to save bandwidth C sends to D the digest $H_p(v)$ instead of the message v. Since, D colludes with C, D does not raise alarm and sends $H_p(u \cdot v)$ to the witness W. The witness cannot detect the discrepancy and declares the verification as successful. Hence, PAG does not ensure the obligation-to-forward guarantee, because C does not send v to D and D proves the contrary to W.

Another variant of the duck attack is possible. To this end, we consider a situation where a predecessor node (B) and the receptor node (C) are malicious and collude to save bandwidth (also colored in gray in Fig. 4). Under this setting, we consider the message transmission scenario as depicted in Fig. 4, where two predecessor nodes A and B wish to send the 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

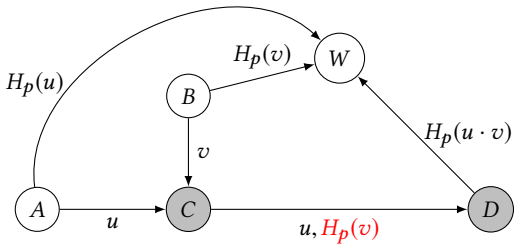

Figure 3: Duck attack on a round of PAG. C instead of forwarding the message v to D, forwards $H_p(v)$. In red, we show the message sent as an adversary.

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, because B never sent to C the message u while D and B prove the contrary to W.

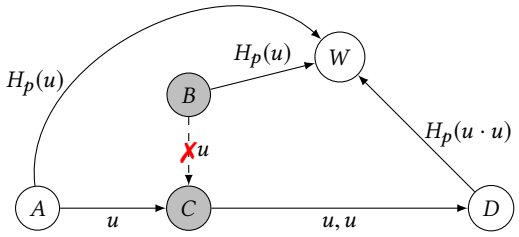

Figure 4: Collusion attack on a round of PAG. Dashed arrow and the symbol ✗ is to denote that this transmission never occurred.

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 [23] is a protocol presented at SRDS 2014 and inspired by PeerReview [16] and FullReview [9]. These protocols rely on maintaining a secure data structure on each node. The data structure implements a *tamper evident log*. The log is used to record the messages a node has sent and received. Eventually, any node (verifier) may request the log of another node (prover) and perform an audit and thereby determine whether the prover has deviated from its expected behavior. In this section, we present an overview of AcTinG and the demonstrate how to mount a duck attack.

We first describe how two nodes interact when a message is exchanged and how the secure-log is built. We then demonstrate our duck attack to this exchange. Next, we show another attack

that allows colluding adversaries to predict when the audits take place. Ability to predict the audits can be exploited by a malicious node to deviate from the protocol in the rounds when it will not get audited.

4.1 Secure-Log in AcTinG

Each node in AcTinG maintains a secure-log that forms a *chained-digest*. The log is an append-only linked list that contains records pertaining to messages sent or received by a node.

The log entry for a node at any given round i is given by $T_i||i||m_i$ which contains an *authenticator* T_i (later used for audit), a sequence number i (a monotonic counter) and a message m_i sent by it. The notation $||$ denotes message concatenation. The next entry of the log is then given as $T_{i+1}||i+1||m_{i+1}$, where T_{i+1} is computed in the following manner:

$$T_{i+1} = H(T_i||i+1||H(m_{i+1})),$$

with H a cryptographic hash function and T_0 a publicly known value. Note that the authenticator chains all previous messages sent/received 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. Fig. 5 presents the chained-digest for a node.

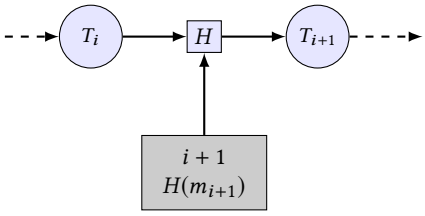

Figure 5: Linear hash chain in AcTinG records each message sent by a node.

Fig. 6 describes how the log entries are created during a communication between two nodes A and B. We assume that A sends a message m to B. In addition to the message m, node A sends $\alpha^A_{i+1} = \sigma(i+1||T^A_{i+1})$, with σ a digital signature algorithm. The value α^A_{i+1} can then be used by any node during the audit process to check the correctness of A. In fact, by sending α^A_{i+1}, A commits to having logged the entry $T^A_{i+1}||i+1||m$ and to the content of its log before it. Any node that receives α^A_{i+1} can use it to inspect the log entry $T^A_{i+1}||i+1||m$ and all the entries preceding it in the log of A.

4.2 Luring AcTinG

We consider the communication scenario of Fig. 6 and assume that nodes A and B are colluding. The duck attack against AcTinG is based on the observation that node B does not need to know m to compute T^B_{j+1} but only $H(m)$ because $T^B_{j+1} = H(T^B_j||j+1||H(m_{j+1}))$ and $m = m_{j+1}$. Since nodes A and B are colluding, it suffices for A to send $H(m)$ to B. Then, B can create a blank entry $T^B_{j+1}||j+1||\emptyset$ which does not contain the message but has a valid authenticator T^B_{j+1} that can be used to compute T^B_{j+2}. Node B can continue to create

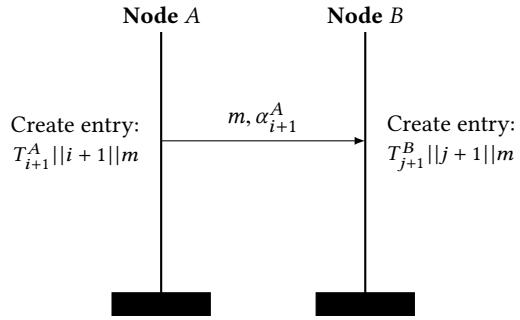

Figure 6: Log entries created by AcTinG during the transmission of m by A to B. We assume that this creates $(i+1)$-th entry in the log of A and $(j+1)$-th entry in the log of B.

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. 7. In the next section, we show how a node may easily determine when it will get audited. This renders the duck attack stealthy and impossible to detect.

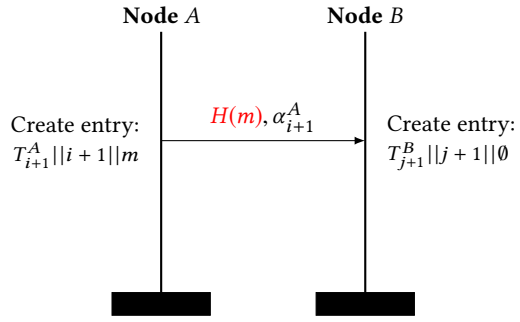

Figure 7: A and B collude to mount the duck attack. Instead of sending the message m, A sends $H(m)$ to B. $H(m)$ suffices for B to generate the correct authenticator T^B_{j+1}.

At 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.

The attack presents an interest even when no compression can be achieved. Node A can pursue other important activities like exchanging other messages with B while still pretending to follow

the protocol. A sends m and m' to B when she is less busy. The only overhead communication overhead is the digests.

4.3 Predicting AcTinG's Audit

In order for the duck attack to go undetected, nodes must exchange the message before the audit takes place. It is hence crucial for the nodes to know when the audit will be performed. In this section, we show how colluding nodes can predict the round in which they will get audited.

The auditing process in AcTinG is as follows. The system ensures that each node gets audited by a set of other nodes. When a node A is associated as an auditor to a node B, A decides if it performs a full audit of B or not. The full audit of B implies checking:

- the secure log of 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.

Auditing in AcTinG is a costly operation and hence is invoked (if at all) at the start of every π rounds (a system parameter). In fact, authors of AcTinG further suggest that the decision to run the audit in a round i (a multiple of π) be taken using a non-deterministic process. However, they provide two deterministic solutions to take this decision:

- Node A chooses a signature $\alpha_j^A = \sigma(j||T_j^A)$ amongst all the signatures in the log after the end of round $(i-1)$ and computes:
$$r = \alpha_j^A \bmod 100$$
- Or, A computes:
$$r = H(PK_A||PK_B||i) \bmod 100,$$
with PK_A, PK_B the respective public keys of A and B.

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: Node A chooses a signature which does not trigger the audit. If the signature is probabilistic (the authors mention in the paper to use RSA 1024 bits signature scheme), the situation is even worse: A can choose the internal randomness used for the signature to make any $\alpha_j^A \bmod 100$ smaller than the threshold and hence 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 [23] too optimistic about the probability to detect a fraud.

5 COUNTERMEASURES

We believe that it is hard if not impossible to find countermeasures for PAG that guarantee both security and privacy. The reasons to this are three-fold: 1) Duck attack is only one of the several possible attacks on PAG. Additional security and privacy issues with PAG are given in Appendix A. 2) There is a lack of well-defined security and privacy properties that a PAG-like system should achieve. PAG only informally presents these properties. Hence, any attempt to fix PAG (if at all possible) would require at the very least well-defined security and privacy definitions. These definitions should be further investigated for any incompatibility. 3) Since the underlying design in PAG is flawed, any possible fix would require building a clean-slate solution which we believe is beyond the scope of this paper. We consider such explorations as important future work.

For the above reasons, we focus on fixing AcTinG. We propose two modifications of AcTinG to defeat the duck attack: employing *verifiable random functions* [10, 13, 14, 22] and hardening the underlying secure-log data structure.

5.1 Using Verifiable Random Function

Recall that in AcTinG, nodes need to use randomness to decide when the audit should be performed and also to choose the nodes they should communicate with. As discussed in Section 4.3, the random number generator as used in AcTinG is either deterministic (and hence predictable) or can be made to output biased coins when used by selfish nodes.

Clearly, randomness is necessary to ensure that malicious or selfish nodes cannot predict the rounds in which they will be audited. Furthermore, one must ensure that malicious nodes abide by the (unbiased) randomness and any attempt to use an alternative source of randomness can be detected. This problem was first tackled by Backes *et al.* in CSAR [3] to extend PeerReview [16] to nodes exhibiting non-deterministic behavior. The underlying idea in CSAR is to use a random number generator that is unpredictable and yet verifiable. This property was achieved using a primitive called *verifiable random functions* (VRF) [22].

Informally speaking, a VRF is a pseudo-random function f_s for a secret seed s such that the owner of the seed can as usual evaluate f_s at any point x but also prove that the obtained value $f_s(x)$ is indeed correct without compromising the unpredictability of f_s at any point x for which no proof of correctness is given. Several constructions of VRFs exist in the literature [10, 13, 14, 22]. For instance, [22] is based on the RSA function, while [10] is based on bilinear maps.

VRFs are the natural option to fix AcTinG. Each node can maintain its own seed s and whenever randomness is required, the node can invoke the VRF f_s on a publicly known input such as a signature or concatenation of public keys (refer to Section 4.3). The output of f_s and the corresponding proof can then be logged to prove to the auditor that the randomness was correctly generated using the function f_s without sacrificing its unpredictability.

Guerraoui *et al.* [15] have proposed a different approach that consists in measuring entropy, but the guarantees are not as strong as those provided by a VRF. Using a VRF ensures that adversaries can not predict or influence the result of the random number generator without being detected.

5.2 Hardening Secure-Log Data Structure

AcTinG [24], PeerReview [16] and several other solutions [3, 9] rely on the secure-log data structure proposed by Maniatis and Baker in [26]. Our attacks described in Section 4 also exploit the same data structure. Hence, in this section, we follow the natural path

to fix AcTinG by modifying the secure-log data structure. To this end, we present a hardening solution motivated by [26]. The key idea is to ensure that the authenticator in round i depends on the current and all previous messages, *i.e.*, $m_i, m_{i-1}, \ldots, m_1$ and not simply on the current message m_i. Additionally, we introduce a final signature to acknowledge reception of a message.

Our approach does not aim to directly detect or prevent duck attacks but to limit its benefit. To this end, we also discuss some limitations and the ensuing security-performance trade-off. The fix presented here is a sketch and we do not provide formal proofs of the guarantees that it yields. This is mainly due to a lack of formally defined security properties against which the efficacy of the fix could be studied.

5.2.1 Adding a Final Signature. Use of digital signatures is a standard way to prevent forgery attacks. A similar modification was proposed in PeerReview [16]. To see how we employ signatures, let us consider two nodes A and B which exchange messages and maintain a log of their communication using our modification (presented in the next section). We 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. 8). 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). Having said that, by introducing a final digital signature, the bandwidth required to mount the duck attack can be increased.

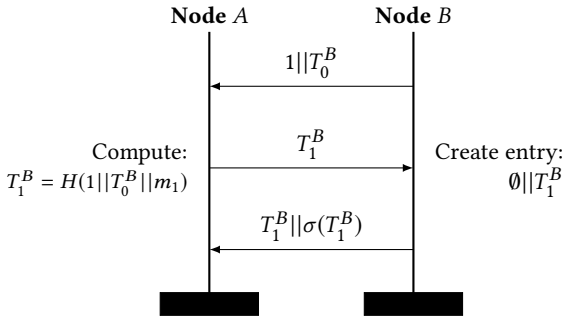

Figure 8: Duck attack on the secure-log structure with a final signature. Introducing a signature as in PeerReview [16] does not help: A and B can exchange values such that B can produce the final signature without knowing the message.

As a side remark, one must note that the duck attack presented above does not apply to PeerReview (which also employs digital signatures) because the protocol does not consider colluding adversaries.

5.2.2 Increased Dependency. We now present the idea of increased message dependency of authenticators. We first look at the secure-log design proposed by Maniatis and Baker in [26]. They in fact present a *secure timeline log*. The security of their design is based on one-way hash function H (instantiated using SHA-1 in 2002[2]). At round 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 as:

$$T_{i+1} = H(i+1||T_i||G(m_{i+1})), \text{ where, } T_0 \text{ is a fixed value.}$$

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 β 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 [26] justified the use of G in their construction by the fact that the m_i can be very large (complete Merkle tree [21] or a large authenticated data structure). To this end, $G(m_i)$ yields a small digest that can be used instead of m_i to reduce the effective size, while maintaining some information about m_i in $G(m_i)$. For instance, in the case of a Merkle tree m_i represents the entire tree, while $G(m_i)$ may represent the root hash of the tree. $G(m_i)$ still helps in ensuring the integrity of the tree leaves.

We modify the above design such that each T_i depends on all previous messages. To this end, we first note that 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 [11]) instead of being only one-way. Our hardened solution modifies the way authenticators are computed. In our modified 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 instead of only m_{i+1} as in AcTinG. Now, for a given $T_{i+1} = H(i+1||T_i||m_{i+1}||m_i||\cdots||m_1))$, it is intractable for an adversary to find $T_i' \neq T_i$ and/or $\alpha \neq m_i$ such that:

$$T_{i+1} = H(i+1||T_i'||m_{i+1}||\alpha||\cdots||m_1)).$$

This is due to the collision resistance property of H. Therefore our modification is secure.

In summary, if we compare the hardened authenticator with the authenticator as used in AcTinG, we have made three changes: 1) A final signature is added at the end of the protocol. 2) The modified authenticator does not depend on the hash of the message (as in AcTinG) but the message itself. 3) It depends on all current and previous messages.

We reiterate that our approach is not to detect or prevent directly the duck attack but to make it more costly than any benefit that two selfish and 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. 9. A and B first execute the duck attack to avoid sending message m_1. Node A computes and sends T_1^B to B. Then, node B receives message m_2 from C. Without the knowledge of m_1, node B cannot compute T_2^B and cannot send the acknowledgement to C. To solve this problem, there are two options for the colluding nodes: A sends m_1 to B or B sends m_2 to A and wait to receive T_2^B. If A sends m_1 to B, they have exchanged more data than if they had followed the protocol. The same conclusion holds if B sends m_2 to A. This problem occurs each time B must log a message from an honest node.

[2]SHA-1 has been recently declared as insecure [27].

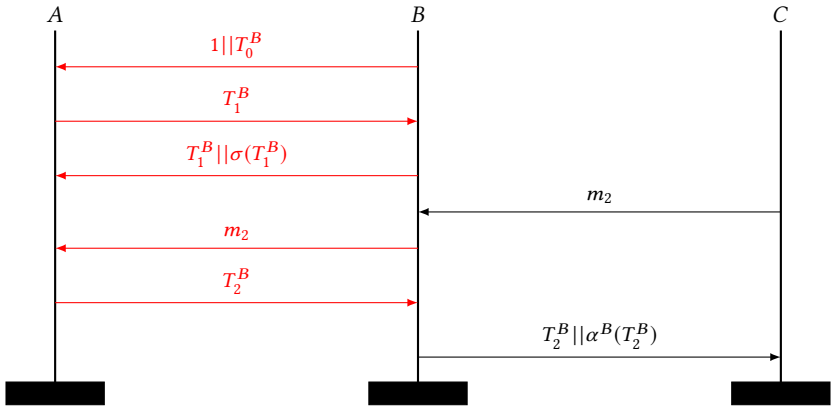

Figure 9: Duck attack when colluding node B receives messages from multiple participants (A and C). Node B must compute $T_2^B = H(2||T_1^B||m_1||m_2)$ but does not know m_1. Only, node A is able to compute T_2^B if B forwards m_2 to A. When A sends T_2^B to B, she can compute $\alpha^B(T_2^B)$ and send the acknowledgement signature to C.

Node B might be tempted to acknowledge the reception of m_2 with a random value $r||\sigma(r)$. However, an audit of B and C will detect this action. Note that maintaining multiple logs per partner is an attack beyond the scope of this paper. This issue has been tackled in [16] for instance.

5.2.3 Security-Performance Trade-off. Fig. 9 clearly shows the benefit of increasing the dependency order of the authenticators. If node B tries to create a blank entry with A, it affects the computation of the next authenticators obtained from an honest node. An issue with our proposed hardening is that the complexity to compute the authenticators in the log grows linearly with the log size. It is possible to find a trade-off between complexity and security by using a window of Δ previous entries:

$$T_{i+1} = \begin{cases} H(i+1||T_i||m_{i+1}||m_i||\cdots||m_{i-\Delta+1}) & \text{if } i \geq \Delta \\ H(i+1||T_i||m_{i+1}||m_i||\cdots||m_1) & \text{if } 0 < i < \Delta \\ H(i+1||T_i||m_{i+1}) & \text{if } i = 0 \end{cases}$$

We have computed the time to create a secure log of 100 entries in `Python 2.7.12` on an Intel Core i7-5500U (2.40GHz) processor . The size of each message is 10KB (arbitrarily chosen). We have used SHA-3 using 512-bit digests. When the computation of authenticators depends on all the previous entries, we observe a slowdown of ×64 compared to the original secure log presented in [26]. Fig. 10 shows that the slowdown grows linearly with the value of Δ. The value $\Delta = 1$ has only a small impact on the performance and can prevent the duck attack if there is a high probability that two consecutive messages are never sent by the same source. This can be achieved by randomly choosing the source in each round.

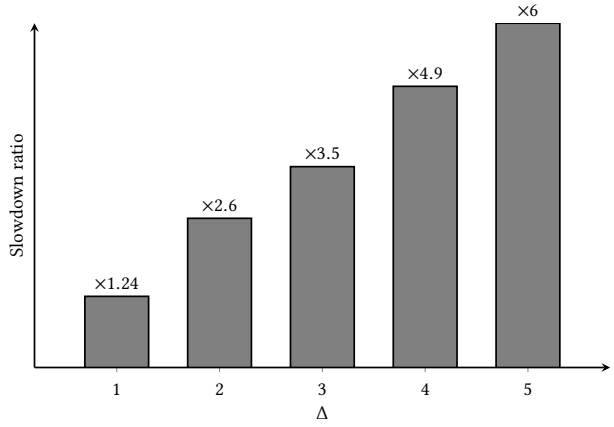

Figure 10: Slowdown observed for the creation of the log when the authenticators depends on the Δ previous entries.

6 CONCLUSION & FUTURE WORK

We have applied the duck attack on a custom cryptographic system PAG and on AcTinG based on secure log. The origin of the attack 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 if not impossible. Designs based on a secure log can however be fixed by increasing the dependency order of the authenticators' computation and by using verifiable random functions.

The proposed fix of AcTinG is a sketch and we do not provide any formal proof of its guarantees. This is due to a lack of formally defined security properties. We consider formally defining the security properties and providing formal proofs as important future work.

More generally, a key issue to go forward in the design of accountable distributed protocols is the definition of the selfish adversaries. Recently in [19] 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

[1] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. 2005. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. ACM, New York, NY, USA, 45–58.

[2] Gildas Avoine, Muhammed Ali Bingöl, Süleyman Kardas, Cédric Lauradoux, and Benjamin Martin. 2011. A framework for analyzing RFID distance bounding protocols. *Journal of Computer Security* 19, 2 (2011), 289–317. https://doi.org/10.3233/JCS-2010-0408

[3] Michael Backes, Peter Druschel, Andreas Haeberlen, and Dominique Unruh. 2009. CSAR: A Practical and Provable Technique to Make Randomized Systems Accountable. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, Virginia, USA, 13.

[4] Elaine B. Barker and Allen L. Roginsky. 2015. *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. Technical Report. National Institute of Standards and Technology (NIST).

[5] Thomas Bonald, Laurent Massoulié, Fabien Mathieu, Diego Perino, and Andrew Twigg. 2008. Epidemic Live Streaming: Optimal Performance Trade-Offs. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*. ACM, New York, NY, USA, 325–336.

[6] S. H. Cavallar, B. Dodson, A. K. Lenstra, W. M. Lioen, P. L. Montgomery, B. Murphy, H. J. J. te Riele, K. Aardal, J. Gilchrist, G. Guillern, P. C. Leyland, J. Marchand, F. Morain, A. Muffet, C. Putnam, C. Putnam, and P. Zimmermann. 2000. Factorization Of A 512-Bit RSA Modulus. In *Advances in Cryptology (Lecture Notes in Computer Science)*, B. Preneel (Ed.), Vol. 1807. Springer, California, USA, 1 – 18. http://oai.cwi.nl/oai/asset/10351/10351D.pdf

[7] Cas J. F. Cremers, Kasper Bonne Rasmussen, and Srdjan Capkun. 2012. Distance Hijacking Attacks on Distance Bounding Protocols. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. IEEE, California, USA, 113–127.

[8] Jeremie Decouchant, Sonia Ben Mokhtar, Albin Petit, and Vivien Quéma. 2016. PAG: Private and Accountable Gossip. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE, Nara, Japan, 35–44.

[9] Amadou Diarra, Sonia Ben Mokhtar, Pierre-Louis Aublin, and Vivien Quéma. 2014. FullReview: Practical Accountability in Presence of Selfish Nodes. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014*. IEEE Computer Society, Nara, Japan, 271–280.

[10] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A Verifiable Random Function with Short Proofs and Keys. In *Proceedings of the 8th International Conference on Theory and Practice in Public Key Cryptography (PKC'05)*. Springer-Verlag, Berlin, Heidelberg, Article 28, 16 pages.

[11] Morris J. Dworkin. 2015. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Technical Report FIPS 202. NIST.

[12] Raphael Eidenbenz, Thomas Locher, and Roger Wattenhofer. 2011. Hidden communication in P2P networks Steganographic handshake and broadcast. In *30th IEEE International Conference on Computer Communications, INFOCOM 2011*. IEEE, Shanghai, China, 954–962.

[13] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. 2015. The Pythia PRF Service. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, Article 35, 16 pages.

[14] Matthew Franklin and Haibin Zhang. 2013. *Unique Ring Signatures: A Practical Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 162–170.

[15] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Swagatika Prusty. 2010. LiFTinG: Lightweight Freerider-Tracking in Gossip. In *Middleware 2010 - ACM/IFIP/USENIX 11th International Middleware Conference (Lecture Notes in Computer Science)*, Vol. 6452. Springer, Bangalore, India, 313–333.

[16] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, Washington, USA, 175–188.

[17] Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. 2003. Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Trans. Parallel Distrib. Syst.* 14, 3 (2003), 248–258.

[18] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. 2010. Factorization of a 768-Bit RSA Modulus. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*. Springer, California, USA, 333–350.

[19] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. 2010. Accountability: definition and relationship to verifiability. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. ACM, Illinois, USA, 526–535.

[20] Harry C. Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Michael Dahlin. 2008. FlightPath: Obedience vs. Choice in Cooperative Services. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*. USENIX Association, San Diego, California, USA, 355–368.

[21] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*. Springer-Verlag, London, UK, UK, Article 32, 10 pages.

[22] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. 1999. Verifiable Random Functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99*. IEEE Computer Society, New York City, USA, 120–130.

[23] Sonia Ben Mokhtar, Jeremie Decouchant, and Vivien Quéma. 2014. AcTinG: Accurate Freerider Tracking in Gossip. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014*. IEEE Computer Society, Nara, Japan, 291–300.

[24] Sonia Ben Mokhtar, Jeremie Decouchant, and Vivien Quéma. 2014. AcTinG: Accurate Freerider Tracking in Gossip. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014*. IEEE, Nara, Japan, 291–300.

[25] João Oliveira, Ítalo S. Cunha, Eliseu C. Miguel, Marcus V. M. Rocha, Alex Borges Vieira, and Sérgio Vale Aguiar Campos. 2013. Can Peer-to-Peer Live Streaming Systems Co-exist With Free Riders?. In *13th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2013, Trento, Italy, September 9-11, 2013, Proceedings*. IEEE, Trento, Italy, 1–5.

[26] Petros Maniatis and Mary Baker. 2002. Secure History Preservation Through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, San Francisco, USA, 297–312.

[27] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. *IACR Cryptology ePrint Archive* 2017 (2017), 190. http://eprint.iacr.org/2017/190

[28] Edmund L. Wong and Lorenzo Alvisi. 2013. What's a little collusion between friends?. In *ACM Symposium on Principles of Distributed Computing, PODC '13*. ACM, Montreal, QC, Canada, 240–249.

[29] Aydan R. Yumerefendi and Jeffrey S. Chase. 2004. Trust but Verify: Accountability for Network Services. In *Proceedings of the 11st ACM SIGOPS European Workshop, Leuven, Belgium, September 19-22, 2004*. ACM, Leuven, Belgium, 37.

[30] Aydan R. Yumerefendi and Jeffrey S. Chase. 2005. The Role of Accountability in Dependable Distributed Systems. In *Proceedings of the First Conference on Hot Topics in System Dependability (HotDep'05)*. USENIX Association, Berkeley, CA, USA, 6.

APPENDIX A: MORE ATTACKS ON PAG

During our analysis of PAG, we discovered several other vulnerabilities. Some of these can be fixed but at the cost of some important properties (privacy for instance). In this Appendix, we present these vulnerabilities and the ensuing attacks.

A.1 Hash Collisions

PAG is also insecure because the cryptographic primitive used to instantiate the function H_p of Section 3.1 has collisions. Recall that H_p is instantiated by a variant of the RSA function with a prime exponent p and a modulus M. For any message $u \in \{0,1\}^*$, the digest of the hash function H_p is given as: $H_p(u) = u^p \bmod M$.

A trivial collision exists for two messages u and v such that $v = u + k \times M$ for any $k \in \mathbb{N}^*$:

$$\begin{aligned} H_p(v) &= v^p \bmod M, \\ &= (u + k \times M)^p \bmod M, \\ &= u^p \bmod M. \\ &= H_p(u). \end{aligned}$$

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 now show how to exploit the hash collisions to mount a *downgrade attack*, where, instead of forwarding a content, a node forwards a downgraded content.

We consider the network depicted in Fig. 11. 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 is used throughout and is known to all the participants of the network (a consequence of our key recovery attack). Also note that, $u' = u \bmod M$, but, u and u' are different messages as PAG assumes that any message exchanged between the nodes is larger than M.

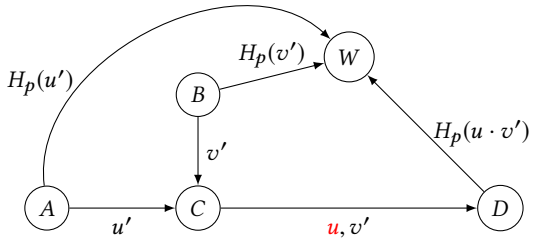

Figure 11: Downgrade attack on a round of PAG. In red, we show the message sent as an adversary.

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

$$\begin{aligned} H_p(u') \cdot H_p(v') &= H_p(u \cdot v') \\ &= H_p(u) \cdot H(v'). \end{aligned}$$

W therefore concludes that C has indeed respected the obligation-to-receive and the obligation-to-forward properties. 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

A strength of PAG is that the node W does not need to know u and v to check that C respects its obligation-to-receive and its obligation-to-forward properties. PAG enforces privacy at the witness level. More concretely, PAG aims to guarantee the following privacy property (taken verbatim from [8]):

Unlinkability between message updates and nodes: *Suppose that node A sends an update u to node B. Nodes other than A and B should not be able to link A or B with u.*

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 key recovery attack that makes the hash 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 [8] 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, φ is the Euler's totient function. Knowledge of $\varphi(M)$ allows to invert any digest $H_p(u)$ and obtain u as:

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

It is to note that an RSA modulus of 512 bits named RSA-155[3] was successfully factored by Herman te Riele *et al.* [6] 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 [18]. In 2015, NIST published a standard [4] 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 linkabiliy 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.

[3] RSA-155 was a part of the RSA Factoring Challenge put forward by RSA Laboratories.