Analysis of a Redactable Signature Scheme on Data with Dependencies

David Bauer
School of ECE
Georgia Institute of Technology
Email: gte810u@mail.gatech.edu

Douglas M. Blough
School of ECE
Georgia Institute of Technology
Email: dblough@ece.gatech.edu

***Abstract*—Storage of personal information by service providers risks privacy loss from data breaches. Our prior work on minimal disclosure credentials presented a mechanism to limit the amount of personal information provided. In that work, personal data was broken into individual claims, which can be released in arbitrary subsets while still being cryptographically verifiable. In applying that work, we encountered the problem of connections between claims, which manifest as disclosure dependencies. In further prior work, we provide an efficient way to provide minimal disclosure, but with cryptographic enforcement of dependencies between claims, as specified by the claims certifier. Now, this work provides security proofs showing that the scheme is secure against forgery and the violation of dependencies in the random oracle model. Additional motivation is provided for a preservation of privacy and security in the standard model.**

I. Introduction

This technical report is provided as a follow-up to our previous work creating a redactable signature scheme that can cryptographically enforce release policies for showing claims with dependencies on other claims. [1] For the convenience of the reader, the system description from (an updated version of) that report is duplicated here, before the security proofs.

A. Scenario and Terminology

We consider a scenario with three types of entities: a prover, a verifier, and a certifier. A prover holds records that are certified (cryptographically signed) by a certifier. The prover wants to convince the verifier that the certifier did indeed certify the records. However, the prover does not wish to release all of the records, but just some subset of them. Additionally, the certifier wishes to restrict the manner in which the records can be released. “Released” here refers only to releasing records with evidence that they are certified (ie, cryptographic proof). The prover can freely forge (uncertified) records, so the verifier will accept only certified documents. We refer to an indivisible piece of a record as a claim, following our earlier credential work.

II. System Description

Our redactable signature with dependencies consists of several parts. The first two parts are the PKI certificate and Merkle hash tree as used in our prior work and described in the previous section. The interesting and novel part is the handling of the dependencies.

A. Dependency Graph

Dependencies between data can come in many forms. The simplest form is a single ”depends upon” relationship, such as ”claim 1 depends upon claim 2”, which means that ”claim 1” should not be released without also releasing ”claim 2”. The next simplest form is a chaining of dependencies, such as ”claim 1 depends upon claim 2, and claim 2 depends upon claim 3”. These chains can be handled by creating one node per claim in the chain, with each node containing its corresponding claim and all subsequent claims in the chain. Less simple is when there are OR options, such as ”claim 1 depends upon either claim 2 or claim 3”. In small numbers, these OR options can be handled by just enumerating the possible combinations as if they were chains, but for large systems, that is extremely inefficient.

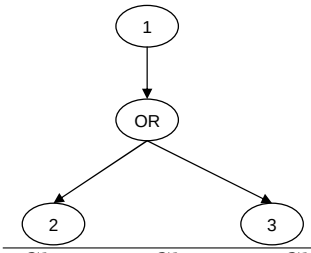
Simple OR dependencies will be represented by a directed acyclic graph (DAG). To handle these dependencies efficiently, a secure hash function is used to create a path whereby a claim is proven valid at the same time as the next node in the graph is proven valid. We call a node that is dependent upon another node a parent of the latter node. The node that is depended upon is called the child node. A node is assigned a ”string” value, which is a hash of the string values of its parent nodes and its actual data value. Calculating a node’s string therefore requires having the data for that node. Just as the node (hash) values in the Merkle hash tree define a unique set of children, each node’s string value defines a unique set of parent nodes and its data value. In order to tie the entire DAG down to a single value, an output value is created, which is simply the set of string values of all of the leaf nodes of the DAG.

Figure 1 shows the notation that we use, while Figure 2 shows the example described above. The example in Figure 3 shows that multiple parent nodes are efficiently handled. In general, the size of the node’s value will grow sub-linearly with the number of parents, as the extra parent strings are amortized by the node’s actual data content.

AND operations are more complex to handle than OR operations. An example including an AND operation is shown in Figure 4. The AND node has two branches for its two children. A different string for the AND is given to each branch, represented by AND1_1 and AND1_2. (The two AND pieces are shown without the S(x) notation, because

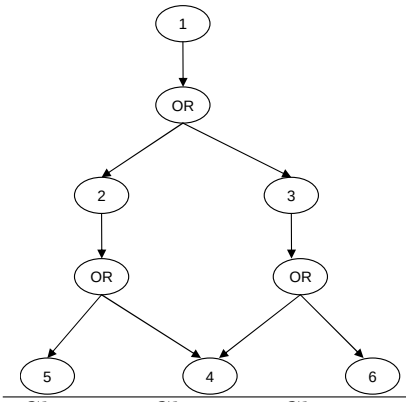
$\rightarrow$ is used for depends upon
$A \rightarrow B$ is read "A depends upon B",
and A is called a parent of B
$+$ indicates concatenation
$\{\}$ indicates a set of values, concatenated together
$S(x)$ is the string for vertex x, and is defined as
$S(x) = H(\{$ parent vertices strings $\} + x)$

Fig. 1. Generic Dependency Form



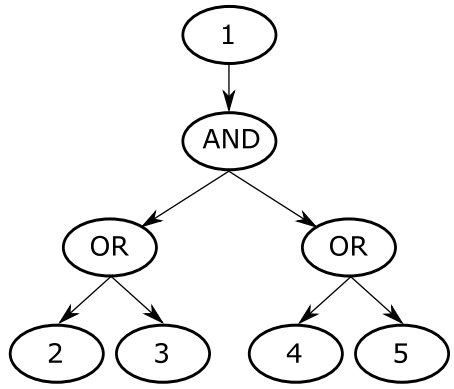
$Claim1 \rightarrow Claim2$ or $Claim3$
$S(Claim1) = H(Claim1)$
$S(Claim2) = H(S(Claim1) + Claim2)$
$S(Claim3) = H(S(Claim1) + Claim3)$
$Output = S(Claim2) + S(Claim3)$

Fig. 2. Simple Dependency Example



$Claim1 \rightarrow Claim2$ or $Claim3$
$Claim2 \rightarrow Claim4$ or $Claim5$
$Claim3 \rightarrow Claim4$ or $Claim6$
$S(Claim1) = H(Claim1)$
$S(Claim2) = H(S(Claim1) + Claim2)$
$S(Claim3) = H(S(Claim1) + Claim3)$
$S(Claim4) = H(S(Claim2) + S(Claim3) + Claim4)$
$S(Claim5) = H(S(Claim2) + Claim5)$
$S(Claim6) = H(S(Claim3) + Claim6)$
$Output = S(Claim4) + S(Claim5) + S(Claim6)$

Fig. 3. Multiple Parents



$Claim1 \rightarrow (Claim2$ or $Claim3)$ and
$(Claim4$ or $Claim5)$
Is transformed into:
$Claim1 \rightarrow AND1$
$AND1 \rightarrow OR1$ and $OR2$
$OR1 \rightarrow Claim2$ or $Claim3$
$OR2 \rightarrow Claim4$ or $Claim5$
$S(Claim1) = H(Claim1)$
$S(AND1) = H(S(Claim1) + AND1\_1)$
$AND1\_1 =$ Random value, of size $|H|$
$AND1\_2 = S(AND1)$ xor $AND1\_1$
$S(OR1) = AND1\_1$
$S(OR2) = AND1\_2$
$S(Claim2) = H(S(OR1) + Claim2)$
$= H(AND1\_1 + Claim2)$
$S(Claim3) = H(S(OR1) + Claim3)$
$= H(AND1\_1 + Claim3)$
$S(Claim4) = H(S(OR2) + Claim4)$
$= H(AND1\_2 + Claim4)$
$S(Claim5) = H(S(OR2) + Claim5)$
$= H(AND1\_2 + Claim5)$
$Output = S(Claim2) + S(Claim3) +$
$S(Claim4) + S(Claim5)$

Fig. 4. Combining AND and OR

they aren't actual vertex nodes in the graph.) When the two branches are XORed together, the result is the actual value of the AND node. For an n-input AND, this is done by generating n-1 random values (each the size of the output of the hash function) and using them as the values for the first n-1 branches. The final branch is the XOR of the rest of the branches and the AND node's value. All of the randomly generated values are included in the string that is hashed to get the AND node's value (to prevent any linear combination attacks against the XOR combination). This is a simple (n, n) secret sharing scheme, which requires the strings of all of the AND node's children to be known to reconstruct the value of the AND node itself. The example also shows how the OR nodes disappear, because their value is equal to their parents' value.

As an example of requiring combined AND and OR dependencies, imagine a table of claims, with each column

Prover provides:
$Output = S(Claim2) + S(Claim3) + S(Claim4) + S(Claim5)$
$S(Claim2) = H(AND1\_1 + Claim2)$
$S(Claim4) = H(AND1\_2 + Claim4)$
$S(AND1) = H(S(Claim1) + AND1\_1)$
$S(Claim1) = H(Claim1)$

Verifier checks:
$Output$ is signed (in the hash tree)
All hash values are correct
$S(AND1) = AND1\_1$ xor $AND1\_2$)

Fig. 5. Showing claims 1, 2, and 4

containing a different type of claim. Consider the rule that to access an element of the first column requires also showing (at least) one element of every other column. Using our method, this requires a graph containing one node for every element in the table, a single AND node, and one OR node for each column but the first two.

Certain dependencies are not handled by our current approach, including some cyclic dependencies, negative dependencies, and operations that cannot be represented as a combination of ANDs and ORs. We do not believe that negative dependencies between released claims is meaningful, since the prover could always perform multiple, independent showings of the signed documents. Cycles are, in general, prohibited, although we handle simple cycles that are not interdependent on other sets of claims by collapsing the entire cycle into a single claim.

B. Protocols for Usage

In general, a set of claims will have some claims with no dependencies and other groups of claims that are inter-dependent. To handle this situation, we combine the structures described in the previous subsection with the hash-tree-based redactable signature scheme from our prior work. Each group of inter-dependent claims is represented by a DAG and a single signed output value is generated for each such group. Each of these signed output values then becomes a node in the overall hash tree, along with each of the claims that have no dependencies associated with them. As in the prior approach, the certifier signs the root value of this hash tree and places it in a PKI certificate.

To show a claim that has dependencies requires showing more claims to fulfill those dependencies. We refer to a claim and one set of additional claims that fulfills the dependencies as a chain. The term "chain" is not strictly accurate, since the chain will have multiple branches if it has any AND nodes, and those multiple branches may even connect together (ie, not a tree). There can be no loops, per the constraint that dependencies must be in the form of a DAG.

As an example, consider the graph of Figure 4 and the case of showing "Claim1", "Claim3", and "Claim4". The prover must provide to the verifier the input strings that were hashed to create the string values of each of the claims being shown and the AND node on the path, along with the (signed) output value. The input string for a claim node includes the actual data of that claim, so the data for the three claims is included in what is shown. Figure 5 summarizes what the prover shows and what the verifier needs to verify. The only additional values given to the verifier that are not used are $S(Claim4)$ and $S(Claim5)$. These are the string values for the other two leaf nodes, and contain just the hash output. Under the assumption that the hash function is secure (can't be inverted and doesn't leak data), then no data is leaked by these extra strings, except for the knowledge of their existence.

III. Security

A. Threat model

The primary threat our system is designed to resist is the prover and verifier collaborating to cheat the certifier, by violating the dependencies on showing claims. In this case, security is done on a "can prove" basis, where it doesn't matter how the certified data is proven. In particular, the verifier does not have to follow established protocols or intentions. (Put simply, we do not assume that the verifier is an honest player in the system.) We refer to the verifier as suspicious (of the prover), but rule-breaking.

B. Definitions

Definition 1. A *node* is an unambiguously parseable set of hash values plus a single claim. Each hash value is the hash of another node in the structure or an AND node secret share.

Definition 2. A node is a *leaf node* if it is contained in the specially defined list of leaf nodes. If a leaf node's hash value can be found in any node in the structure, the entire structure must be considered invalid by the verifier.

Definition 3. A non-AND node is *accepted* by a verifier if it is in the list of leaf nodes, or if its hash value is found in an already accepted node. An AND node is *accepted* if its hash is created by combining shares from already accepted nodes. A claim is *accepted* if it is in an accepted node.

Definition 4. The *directed acyclic graph (DAG)* consists of all nodes reachable when starting from the list of leaf nodes.

Definition 5. A task is called *possible* if and only if it is computationally feasible.

C. Forgery

Forgery covers all cases where the verifier is convinced that a claim is certified by a particular entity, when it was not. Forgery covers several different problems, depending on what part of the system is attacked. For example, a forger can try to attack the hash function to create a bad final or intermediate value. A secure hash function will prevent this type of attack.

Overview: the structure of the system is a signed DAG. It is shown that a verifier will only accept data if it was in the DAG when the DAG was created/signed and that a creator will only sign a DAG when all data in it is known.

Theorem 1. *The verifier will only accept as valid data items that were known to be in the structure by its creator.*

1) Assumptions:

- A collision-resistant hash function is used.
- The set of leaf nodes is fixed via a digital signature. The use of a redactable signature, such that not all leaf nodes are known to the verifier has no impact on our proof.
- The structure and type of a node is known and unambiguously parseable.
- The certifier will not put in hash values of nodes for which the contents are unknown.
- The certifier will not put a malformed node into the DAG.
- The "random" values used for AND nodes are generated by the certifier (and are not accepted from another entity).
- The hash function may be modeled as a random oracle [2] (for AND nodes).

2) Without ANDs: *Proof:* First, by the collision resistance assumption on the hash function, it is not possible to create two nodes with the same hash value but different contents. By definition, a leaf node will be accepted if and only if its hash is in the list of leaf nodes and a non-leaf node will be accepted if and only if its hash is given in an already accepted node. By induction, a node will be accepted if and only if it is either in the list of leaf nodes or connected to a node in the list of leaf nodes via a chain of accepted nodes, i.e., it is in the DAG.

A node's value and therefore hash depends on the hash values of the nodes to which it links and the claim it contains. Therefore, a node's value cannot be computed without the hash of the nodes to which it links already being known. By induction, a node's value cannot be computed without the hash of all nodes 'down' from it in the DAG being known. Since this applies to all nodes, it applies to the leaf nodes. Thus, the set of leaf nodes cannot be computed without knowing the hash values of all other nodes in the DAG. By assumption, the creator of the structure will not accept the hash value of a node without also knowing the contents. Thus, since the hash values of all nodes in the DAG must be known, the contents of all nodes in the DAG must be known. And since the verifier will only accept nodes in the DAG, the verifier will only accept nodes known to be in the structure at creation time, and thus data items that were in the structure at the time of its creation. ■

3) With ANDs:

Lemma 1. *Given a secure hash function H that may be modeled as a random oracle, a list of values $\{x\}$ in the domain of H such that $|\{x\}| <<< |H|$, it is not computationally feasible to find a set of values $\{y\}$ such that $x' = H(\{y\})$ xor $\{y\}$, where x' is any element of $\{x\}$.*

Proof: $H(\{y\})$ is a completely different random value for each different $\{y\}$. XORing in non-random (but chosen ahead of time) values can't make the output any less random. Thus, $H(\{y\})$ xor $\{y\}$ is still a random value for each different $\{y\}$. The probability of such a random value appearing in $\{x\}$ is $|\{x\}|/|H|$, which is very small in a practical system. ■

Proof: By the assumptions of a secure hash function and a reasonable generation of (pseudo) random values for AND nodes, it is not possible to find a node that hashes to any of the "random" values. Similarly, it is not possible to find a node that hashes to the parity value for an AND node. Thus, it is not possible to fake a non-AND node from the values and constructions provided for AND nodes. An AND node itself has a specific form. By the assumption of unambiguous nodes, an AND node cannot be mistaken for a non-AND node. By the definition for accepting an AND node and the assumption that a certifier will not put a malformed node into the DAG, an AND node is accepted if and only if the self-check values contained in it and the parity value are all given in already accepted nodes. (Since it is trivial to forge an AND node if one of the check values may be freely determined, a sincere verifier cannot accept an unverified/unaccepted check/parity value.) The case of forging an AND node using only accepted values is equivalent to the case given in Lemma 1, but with $\{y\}$ constrained to come from the same list as $\{x\}$. As such, by Lemma 1 it is not possible to forge an AND node from already accepted values. Since it is not possible to forge either an AND or non-AND node using the AND node constructs, the addition of AND nodes does not allow any forgery not already possible. But, it was already shown that in the case with no AND nodes, it is not possible to get any forgery accepted. ■

D. Loss of Privacy

A loss of privacy occurs when the verifier learns something that the prover doesn't mean to show.

In the system, there is only one place where additional information is given out: the hash value of some nodes that the prover doesn't show will still be given to the verifier. This could allow an attacker to perform a dictionary attack against a node, except that random padding makes such an attack infeasible.

E. Violation of Dependencies

A violation of dependencies occurs when the prover can convince a suspicious, but rule-breaking verifier that the creator certified a claim (which is in the structure), without respecting the dependencies that the signer expected to be enforced.

1) Additional Assumptions:

- The "random" values must look random, such that it is not possible to find a (simple) generator that will produce the AND values.

2) Without ANDs: *Proof:* From the previous proof against forgery, it is already known that an attacker cannot get a verifier to accept a node which was not in the DAG when the creator signed it. A node (in the DAG) will not be accepted by a verifier unless the hash of that node is given in an already accepted node, a leaf node, or an accepted partial node. An accepted partial node is the case where a prover can convince a verifier that a bit of data is part of a valid node, without showing the whole node. Under the random oracle model of a

hash function, it should not be possible to convince a verifier to accept any partial node, because

1) The only evidence a verifier should accept is a hash value
2) The hash value can only be computed by the random oracle given the entire node
3) Therefore, a verifier cannot check or accept a partial node

Using a real hash function that processes data such that later data cannot be processed without earlier data (or intermediate values from already processed earlier data), it is easy to see that even using intermediate values from the hash calculation can't allow a prover to make malicious use of partial nodes, because the data that the node is showing is the last part of the node to be hashed. Virtually all hash functions meet this criteria, with the exception being hash functions using tree-based hashing (such as MD6 [3] or the optional mode of Skein [4]). With partial nodes excluded, only hash values in already accepted nodes or leaf nodes will be accepted by a verifier as evidence of a node belonging in the DAG. Additionally, since such accepted nodes or leaf nodes cannot be accepted by the verifier without knowing their contents, the dependencies upon them are maintained. ■

3) With ANDs: *Proof:* The new assumption is necessary to prevent a prover from convincing the verifier to accept a "random" value for use in an AND node that has not been shown in an already accepted or leaf node. If the values chosen form an obvious pattern, or are generated by a weak pseudo-random number generator, the verifier may accept a confirmed pattern as sufficient evidence to accept a value.

Under this random-looking assumption, the verifier will only accept "random" values that appear in already shown and accepted nodes or leaf nodes. By the assumption that the certifier generates the "random" values and the unforgability of AND nodes, those "random" values will be values intended as random values by the prover. It was shown above that in this situation, the verifier can only show nodes in the DAG. As the verifier cannot know that a node is in the DAG without the full path going back to leaf nodes, the difficulty of violating the dependencies reduces to the difficulty of having a forged node accepted by the verifier or getting a partial node accepted. Both of these were already shown to not be possible. ■

IV. Acknowledgments

This research was supported in part by the National Science Foundation (under Grant CNS-CT-0716252) and the Institute for Information Infrastructure Protection. This material is based in part upon work supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of any of the sponsors.

References

[1] D. Bauer, D. M. Blough, and A. Mohan, "Redactable signatures on data with dependencies," CERCS, Georgia Institute of Technology, Tech. Rep., 2009. [Online]. Available: http://www.cercs.gatech.edu/tech-reports/tr2009/abstracts/03.html

[2] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*. New York, NY, USA: ACM, 1993, pp. 62–73.

[3] R. L. Rivest, "The md6 hash function – a proposal to nist for sha-3," Submission to NIST, 2009. [Online]. Available: http://groups.csail.mit.edu/cis/md6/docs/2009-04-15-md6-report.pdf

[4] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The skein hash function family," Submission to NIST, 2008. [Online]. Available: http://www.schneier.com/skein.pdf