Redactable Signatures on Data with Dependencies and their Application to Personal Health Records

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

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

Apurva Mohan
Georgia Institute of Technology
School of ECE
apurva@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 control the dissemination of personal information. 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 this work, we provide an efficient way to provide minimal disclosure, but with cryptographic enforcement of dependencies between claims, as specified by the claims certifier. This provides a mechanism for redactable signatures on data with disclosure dependencies. We show that an implementation of our scheme can verify thousands of dependent claims in tens of milliseconds. We also describe ongoing work in which the approach is being used within a larger system for dispensing personal health records.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Authentication

General Terms

Algorithms, Performance, Design, Security, Verification

1. INTRODUCTION

The amount of personal information that is supplied by individuals and stored electronically by other entities, primarily service providers, is enormous and growing. This information ranges from basic (and yet still sensitive) attributes such as name, address, date of birth, and social security number to payment information, e.g. credit card numbers and expiration dates, to much more comprehensive personal information such as detailed financial records and medical records. Unauthorized disclosure of this personal information is a major problem that has been steadily increasing in severity over the last several years.

The overarching goal of our research is to give users more control over their personal information, while also providing trust in the information that is supplied. An important component of our approach is the principle of "least disclosure", i.e. that an entity requesting personal information should be given the minimum amount of information required to authorize the necessary operation or transaction. Our prior work developed new redactable signature schemes that were used to design *minimal disclosure digital credentials*, which combine a large set of attributes into a digital credential with one signature, which can be used to verify any subset of the attributes [1]. As opposed to a single "all-or-nothing" credential, this allows the user flexibility to provide some attributes, while hiding the remaining ones, but it still allows the credential recipient to verify cryptographically and efficiently the provided attributes.

While digital credentials are one application area for minimal disclosure technology, it can also be applied to general disclosure of sensitive personal information, e.g. financial records or medical records. Medical record protection is the subject of the MedVault project [6], which is a joint undertaking between Georgia Tech and Children's Healthcare of Atlanta. Each individual element of personal information that can be disclosed is referred to as a claim. In medical records, a claim is an individual component of the record, e.g. an X-ray image or doctor's notes about an office visit. In this paper, we consider how to apply minimal disclosure technology in the form of redactable signatures to the problem of selective disclosure of medical record claims.

Recently, due to entry into the field by major software companies,[1] personal health records, or PHRs, have received a great deal of attention. One major limitation of existing PHR repositories is that there is no way for health care providers to export medical record information to a patient's PHR repository and then for the patient to disclose that information to third parties in a selective and verifiable manner. By verifiable, we mean that the third party should be able to verify that the information came from a particular health care provider and that it has not been modified. By selective, we mean that the patient should control exactly which claims are released from the overall record to a particular third party. In this paper, and as part of our MedVault project, we propose to use redactable signatures as an efficient way to permit health care providers to export signed medical record information to PHRs, while allowing the patient to selectively disclose the claims in the record.

[1]See Microsoft HealthVault [8] and Google Health [3]

WPES'09, November 9, 2009, Chicago, Illinois, USA.

As described so far, a standard redactable signature scheme would suffice for a verifiable PHR repository. Health care providers would sign medical records using a single redactable signature and patients would disclose arbitrary subsets of the medical claims to third parties. However, it is unlikely that health care providers would cede complete control of how the medical claims they produce are disclosed by the patient. For example, a health care provider might not be willing to release doctor's notes containing a medical diagnosis unless the test results on which the diagnosis is based are also released. As a second example, the provider might not be willing to release an X-ray image without also releasing meta-data about the image such as when it was taken, which part of the body it corresponds to, and perhaps notes detailing the provider's interpretation of the image. For convenience in searching and browsing, however, the meta-data should be available without the image.

The preceding examples introduce dependencies between claims, which should limit what combinations of claims can be disclosed. In Section 4, we describe how to handle a certain class of these disclosure dependencies by generalizing a hash-tree-based redactable signature scheme. In Section 5, we discuss the security of our approach. In Section 6, we evaluate both its worst-case complexity, through analytical evaluation, and its execution time, through measurement on an actual implementation. The results show that the efficiency of simple hash-tree-based redactable signatures can be maintained while handling a useful category of claim dependencies. Finally, in Section 7, we describe an architecture for use of our mechanism in a source-verifiable selective-disclosure personal health record repository, which we have developed as part of our MedVault project.

2. BACKGROUND

2.1 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.

2.2 Redactable Signature Schemes

This problem arose out of our previous work on minimal disclosure credentials [1]. In that work, we presented a credential system based on Merkle hash trees and public-key infrastructure (PKI) certificates, which allows some attribute values to be hidden on a given use of the credential and which is essentially equivalent (modulo implementation details) to the redactable signature scheme of [5]. We also extended the approach to allow attributes certified by different identity providers to be combined into a single credential, while still allowing the selective disclosure of attributes in the credential. In this paper, we describe only the basic scheme; the multiple-authority scenario is not considered.

A Merkle hash tree is a binary tree where each internal node holds the hash of the concatenated values of its two children nodes. Ralph Merkle first introduced this structure as a way to efficiently handle a large number of Lamport one-time signatures[7].

By the collision-resistance property of a cryptographic hash function, it should not be possible to find two inputs that give the same output (under reasonable computational limits, of course). Therefore, each internal node uniquely fixes the values of its two children nodes. Extrapolating, the single root value uniquely fixes the value of all internal and leaf nodes in the tree. Additionally, it is not necessary to have all nodes of the tree in order to compute its root value. Specifically, verifying that a given value for a leaf node of the tree is correct requires an additional number of nodes equal to the height of the tree minus one ($\mathbf{O}(log(n))$ with respect to the size of the tree). Having a trusted authority digitally sign the root value of the tree provides the equivalent of a trusted signature on everything in the tree. Replacing a simple digital signature with a full PKI certificate containing a public key and meta-data ties the contents of the tree to a private key through a trusted signature; in short, a digital credential. Each leaf node in the tree is a claim of a particular attribute value, which can be verified independently of all of the other claims in the tree.

2.3 Motivation for Dependencies

From a basic credential system, we expanded the scope of our uses for the credential construct to hold arbitrary records. In doing so, the issue of dependencies between different data items or between different whole records arose. We wish to provide a way for the certifier of data to prevent data from being disclosed without respecting the relationships between different pieces of data. As mentioned previously, "disclosed" refers only to releasing the data in a certified form. In this work, relationships between data are reduced to dependencies between data items that must be satisfied for the data to be disclosed.

3. RELATED WORK

While we are not aware of any other attempts to address this specific problem of dependencies in releasing certified data, this work is still related to various prior works.

The most closely related works using graphs or circuits of dependencies involve fulfilling dependencies under a significantly different threat model, for example [4]. In our system, the most difficult threat is the prover and verifier collaborating to cheat the certifier. And the threshold of that cheating is low, since the result only has to be trusted by the verifier, and not by any honest party.

This work can be considered the core of a specialized redactable signature scheme, although it is meant to be embedded into a separate redactable signature of the hash-tree type described in the previous section. Redactable signatures were introduced by Johnson, et al., in [5] as one example of a larger class of homomorphic signatures. While the targeted use of the schemes is very different—Johnson, et al., describe a scenario where the majority of a document is shown, with a small part redacted, while our work describes showing a small amount of data and redacting the majority—the core mechanism is the same.

Privacy preserving trust negotiations associated with the disclosure of sensitive attributes also have disclosure depen-

dencies among attributes [15, 12, 11, 16]. A user can set up policies where the other negotiating party has to disclose some attributes before a particular attribute of the user can be released. One major difference between trust negotiation systems and our proposed system is that the trust negotiation systems are online approaches, i.e. they expect the other party to release credentials online to satisfy the dependencies as opposed to our offline system where the dependencies are enforced cryptographically. Another difference is that in trust negotiation systems, both parties try to satisfy each other's policies at run time, whereas in our system the policies result from natural dependencies in the data set that must be preserved to release the information in the correct context. Finally, in trust negotiation, each party trusts its negotiator to faithfully execute its own policies, whereas the party releasing information in our approach is not fully trusted by the source of the information and the releasing party cannot therefore be relied upon to faithfully execute the source's policies.

4. 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 Section 2. The interesting and novel part is the handling of the dependencies.

4.1 Dependency Graph

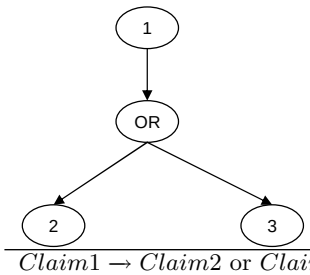
Dependencies between data can come in many forms. The simplest form is a single "depends upon" relationship, such as "claim 1 depends upon claim 2", which means that "claim 1" should not be released without also releasing "claim 2". The next simplest form is a chaining of dependencies, such as "claim 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.

We represent simple OR dependencies 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. The hashed value of the node for Claim 4 simply has two parent strings instead one. In general, the size of the node's value will grow sub-linearly with the number of parents, as the extra parent strings (each just a single hash value) are amortized by the node's actual data content.

$\rightarrow$ 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 $\} + x)$

Figure 1: Generic Dependency Form



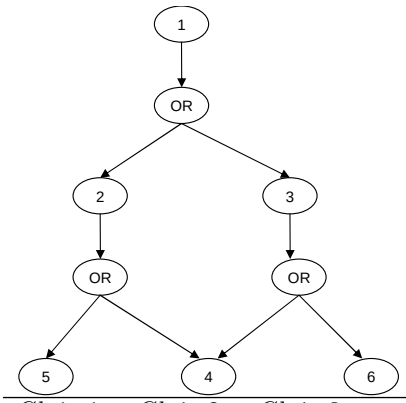
$Claim1 \rightarrow Claim2$ or $Claim3$
$S(Claim1) = H(Claim1)$
$S(Claim2) = H(S(Claim1) + Claim2)$
$S(Claim3) = H(S(Claim1) + Claim3)$
$Output = S(Claim2) + S(Claim3)$

Figure 2: Simple Dependency Example

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 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.

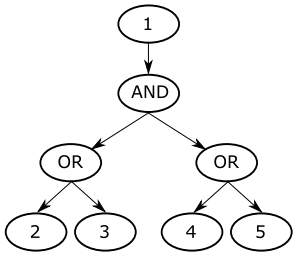
As an example of requiring combined AND and OR dependencies, imagine a table of claims, with each column containing a different type of claim. Consider the rule that to access an element of the first column requires also showing (at least) one element of every other column. Using our method, this requires 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 cyclic dependencies, negative dependencies, and operations that cannot be represented as a com-



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

Figure 3: Example Showing 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 $AND1\_1$
$S(OR1) = AND1\_1$
$S(OR2) = AND1\_2$
$S(Claim2) = H(S(OR1) + Claim2)$
$= H(AND1\_1 + Claim2)$
$S(Claim3) = H(S(OR1) + Claim3)$
$= H(AND1\_1 + Claim3)$
$S(Claim4) = H(S(OR2) + Claim4)$
$= H(AND1\_2 + Claim4)$
$S(Claim5) = H(S(OR2) + Claim5)$
$= H(AND1\_2 + Claim5)$
$Output = S(Claim2) + S(Claim3) + S(Claim4) + S(Claim5)$

Figure 4: Combination of AND and OR

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 $AND1\_2)$

Figure 5: Showing claims 1, 2, and 4

bination 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 prohibited because it is impossible to calculate the values for the nodes in the cycle without breaking the hash function used. Our implementation detects cycles using Tarjan's algorithm [13] and combines all nodes in each strongly connected component (i.e., all nodes involved in a cycle) into a single node (per component).

4.2 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", "Claim2", 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 $S(Claim3)$ 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.

4.3 Relation to prior work

The mechanism for handling dependencies obviously shows a family relationship to the Merkle hash tree. It in fact looks like a backward tree, where instead of a root node verifying the values of many leaf nodes, a set of leaf nodes can verify the value of a set of root nodes (and all of the intermediate nodes as well). In most of our examples, the set of leaf nodes is much larger than the set of root nodes, but that is simply selection bias of the examples.

5. SECURITY

5.1 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.

Formal security proofs for the dependency violation attack as well as for forgery attacks and privacy violation attacks are contained in [2]. Only informal security arguments are given herein due to length restrictions.

5.2 Additional details

A secure scheme requires some additional details beyond those described so far. First and foremost, the claims must have random padding to prevent a dictionary attack, because the hash value of unreleased claims is necessarily provided to a verifier. Second, and related, is that nodes should be unambiguous about what they contain. Throughout this paper, the value of nodes is represented by a simple concatenation of values. In our actual implementation, we include additional meta-data identifying the node type and the length of fields. Additional meta-data might also be required in specific applications of the approach. For example, if applied to documents, the order of the claims (words) is important and this information can be provided in the meta-data in the form of sequence numbers. If knowing the sizes of the gaps in the provided text is important, contiguous sequence numbers are used. Otherwise, random increments between consecutive sequence numbers can avoid release of extra information such as the precise number of words that were redacted from a particular section.

5.3 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.

A forger can try to pass off data as if it were part of the structure, or vice versa. This style of attack is only possible if the construction of nodes' strings is done in a simplistic manner, such as merely concatenating the values. Because of the additional meta-data, discussed above, this attack is not possible.

A forger could also try to fake a valid looking AND node. The AND node construct of XORed masks is a trivial secret sharing scheme, while having the masks inside of the hashed string is a constraint on the values of shares that are accepted. This constraint is necessary to eliminate the possibility of generalized birthday attacks [14]. The problem of faking an AND node can be reduced to the problem of creating a collision in a hash function defined as $H(x + \{y_i\}) = H'(x + \{y_i\})$ xor $\{y_i\}$, where $H'(x)$ is a hash function (assumed to be secure) and $\{y_i\}$ is an arbitrarily sized set of values. This construct is easily shown to be secure when $H'(x)$ is modeled as a random oracle, and we believe it to be secure for practical instantiations of $H'(x)$

5.4 Loss of Privacy

A loss of privacy occurs when releasing some claims or data exposes additional claims or data that the holder did not intend to release. In our system, the most obvious way for additional data to leak is through hashes of unreleased data being (necessarily) revealed. Under the assumption of a secure hash function, no information about the data should be directly leaked by its hash value. However, if the data itself is in a guessable form and of low entropy, then a dictionary attack may be performed against the hash. Simply adding random padding fixes the problem. The padding can either be independently generated and stored for each data value, or it can be generated using a pseudorandom function along with a random seed value.

5.5 Violation of Dependencies

A violation of dependencies occurs when a prover is able to release certified data to a verifier in a way that the verifier can determine that it was certified, while not releasing other data as required by the certifier.

Dependencies are enforced by providing chains, such that each link in the chain must be fully verifiable in order for the next link to be verifiable. Additionally, to verify a (data) link requires the actual data for that node to be known and used by the verifier. These chains overlap, creating the DAG. There are two general ways to violate the dependencies: forging a link in the chain and finding a different way (than following the chain) to prove a claim. The first of those is covered by the subsection on forgery, discussed previously. The second is beyond the scope of this paper, as it covers application-specific side-channels, and is dependent upon the implementation and use of the system.

5.6 Other

There is an additional class of attacks that are of less concern, namely attacks by the certifier of the data. The only meaningful attack we can see is the certifier putting a hidden channel into the signature, without the knowledge of the user. As we generally expect the certifier to provide the data, define the dependencies, and create the actual structures of the signature, we consider attempting to identity and prevent all possible hidden channels as beyond the scope of this paper. Since in all applications we can imagine for the approach, the user has an inherent trust relationship with the certifier, we do not consider this issue a serious one.

6. PERFORMANCE RESULTS

Good performance for large systems of dependencies was a primary goal of this work. Two evaluations of the scheme in meeting that goal are provided. The first evaluation is in the form of analytical bounds, while the second is based on experimental results from an implementation of the scheme.

6.1 Analytical Bounds

Our scheme provides clear bounds on space and time complexity. There is exactly one vertex node for each claim, and the actual data of each claim is stored and hashed only once. There is a maximum of one vertex for each AND and OR as well (multiple ANDs and ORs may be combined).

For comparison, consider a brute-force approach to handling dependencies. All possible combinations of claims necessary to satisfy a particular claim's release policy can be combined and treated as a single claim. For small graphs, this can actually be quite efficient. However, consider the example given before of a table of claims, with the rule that to access an element of the first column requires also showing (at least) one element of every other column. As noted before, the number of nodes in the graph of this example is equal to the number of items in the table plus one (ignoring the removed OR nodes). Assuming a symmetric table (all columns have the same number of rows), the approximate total size of data being hashed in the creation of the graph is given in Equation 1, where n is the number of rows, k is the number of columns, $|H|$ is the size of a hash:

$$(nk + n + k - 3) \cdot |H| + |\text{all data items}| \quad (1)$$

Therefore, our solution is $\mathbf{O}(nk)$ in space and time, both for dependency graph construction and for claim verification. For comparison, enumerating all possible claim combinations is $\Theta(n^k)$ in this example.

Later, we discuss an input graph based on dependencies between software packages available in the Ubuntu software repositories. Tests are done on a graph of approximately 45,000 nodes (holding about 25,000 claims). There are more than six billion possible path combinations of released claims that would have to be separately calculated and stored under the brute-force approach.

6.2 Trade-offs

There are a number of trade-offs that can be made between storage space and speed. For example, separate, smaller DAGs can be calculated and stored for every claim that is involved in a dependency calculation, allowing for quick retrieval and verification of single claims and verification chains. Alternately, a single large DAG (or a small set of medium sized DAGs) can be used, which can greatly increase the amount of data necessary to verify a single claim (and chain), but which is much more efficient to store and verify large sets of claims (with overlapping dependency chains).

The discussion so far has assumed a single, monolithic DAG is used. A single monolithic graph is easier to describe, program, and analyze. However, a multiple graph approach can produce better results in a "best-case" scenario. As such, both a monolithic version and a multiple graph version were implemented for experimental performance testing. To provide the best contrast, the multiple graph version makes and stores a separate DAG for every claim.

6.3 Experimental Setup

A Java implementation of the dependency handling portion of the system was evaluated. Tests were performed on a Dell PowerEdge 2900 server with dual Xeon 5150 CPUs running at 2.66 GHz using Sun's 64-bit server JVM, version 1.6.0_06. The code was single threaded and the server otherwise idle. The minimum time seen over multiple rounds was recorded, to avoid artifacts from just-in-time compilation and garbage collection. SHA-256 was used for the hash function [9].

Two styles of artificial input dependency graphs were used, both rectangular tables. Using this format, it was easy to vary the number of rows, number of columns, and size of data in each claim (constant across all claims in a table to make the results more consistent). The first style matched the example analyzed earlier, where a table of claims has the requirement that to release any claim in the first column requires releasing (at least) one value from each of the other columns as well. This graph is fast to process, as would be expected from the earlier bounds discussion and some simple analysis (most of the nodes in the graph are of low degree, with the connections concentrated in the AND and OR nodes).

The second graph is a table where each column is dependent upon the next column, i.e. showing a claim in any column except the final column requires showing at least one claim from the next column. (For claims in the first column of the table, the behavior of this graph style is the same as for the previous graph style.) This graph is denser in connections than the first graph, with all nodes being of degree n or $2n$.

One large, real-world dataset was tested. This dataset was derived from the dependencies between software packages in the standard Ubuntu Linux repositories. The "Depends" and "Pre-Depends" values were used to determine the dependency graph; other package links ("Recommends", "Suggests", "Replaces", etc) and version information was ignored. A total of 25157 packages were used, representing 21GB of data. Fake empty files of the correct size for each package were created, stored as sparse files on an ext3 formated partition. (The file system reports only 820KB of actual disk space used.) The dependency graph contains 566 cycles containing 2977 nodes, which are condensed down to 566 new nodes. Timings were taken using both short strings, consisting of the package name plus 18 fixed characters, and the full length fake package files.

6.4 Experimental Results

6.4.1 Absolute timings

Handling of dependencies is efficient for most cases, as can be seen in Table 1, which shows timing data for the first graph style. Most of the operations take at most 10 milliseconds. However, the multiple graph approach can be inefficient when the number of claims to be verified is very large. For example, for a 64×128 table with 100 bytes per entry, the multiple graph approach requires 5 seconds to verify *all* claims, while the monolithic graph approach only takes 74 milliseconds for the same case.

The second graph style has the same complexity bound as the first style, but is much slower in practice due to the high density of connections. When analyzing this style, we found that the time to pre-generate all of the chains in the multiple graph approach makes it impractical. Thus, we

Input Table Size			Monolithic Graph		Multiple Graph	
Rows	Columns	Data size	Verify chain	Verify all	Verify chain	Verify all
Small inputs						
4	4	10	360	330	120	450
4	8	10	520	460	200	660
4	16	10	960	890	400	1500
4	32	10	1900	1800	950	3600
Medium inputs						
64	16	100	1700	8200	1300	77,000
64	32	100	3400	17,000	4400	280,000
64	64	100	6800	34,000	19,000	1,200,000
64	128	100	15,000	74,000	77,000	5,000,000

Table 1: Timings for first graph style (in μs)

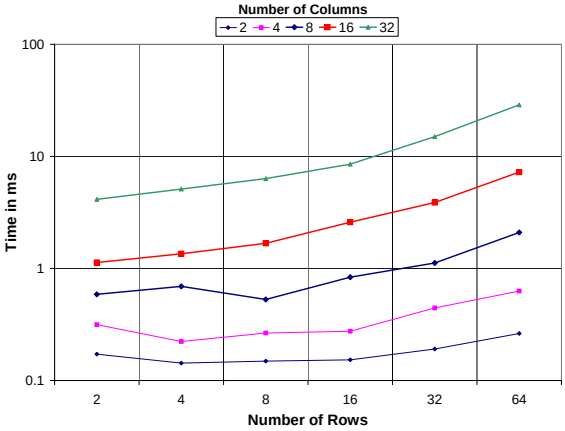


Figure 6: Verifying a Chain in 2nd Graph Style

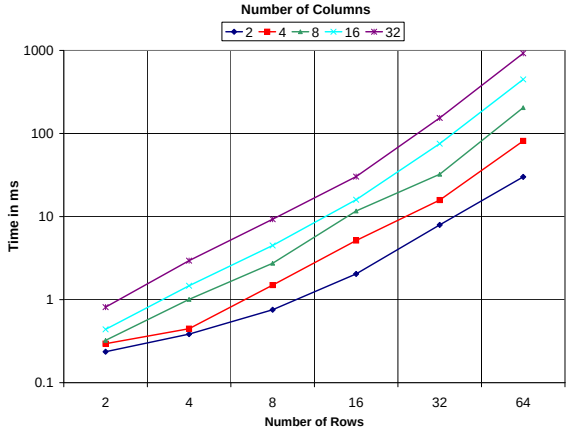


Figure 7: Verifying Full Graph in 2nd Graph Style

report only the monolithic graph results for this case. Figure 6 shows the time (in milliseconds) for verifying a single chain in a monolithic graph (compare to the first timing column in Table 1) using the second graph style. The time is very small (at most tens of milliseconds), even when the numbers of rows and columns become large. Figure 7 shows the time (in milliseconds) for verifying the entire monolithic graph (compare to the second column in Table 1) using the same data. This time is also reasonable, being less than one second even for large cases.

The Ubunutu package dependency graph was only timed using the monolithic graph approach. The timings are shown in Table 2. The difference between the short string and full data versions is just the amount of data in each node that has to be hashed. As can be seen from comparing the two lines in the table, the full data version takes approximately an order of magnitude longer than the short strings version. The additional time is primarily hashing the data. Thus, for the full data version, the overhead of graph manipulation is small compared to the time needed to simply hash the data.

Unlike the two table-based graphs, the length of chains in the Ubunutu package graph varies considerably. The measurements were taken by selecting 2000 pseudorandom chains out of the full graph. The same pseudorandom paths were used in both the short string and full data cases. The average path length was 17 nodes, while the median was 8 nodes.

6.4.2 *Relative timings*

Here, we consider the relative performances of the monolithic and multiple graph approaches. Figure 8 shows the speedup of the multiple graph approach over the monolithic approach. (This graph is based on the first graph style and a data size of 10 bytes.) The multiple graph approach provides a significant advantage—about 35 times faster for the peak in the figure—for some parameter values. In particular, it does best with a large number of rows and a small number of columns. The area to the right of the first contour line is where the monolithic graph performs better. Thus, the optimum implementation approach is dependent on the nature of the dependencies. However, from a worst-case standpoint, the monolithic graph implementation is superior.

7. APPLICATION TO PATIENT MEDICAL DATA

One of the application areas where these ideas are being implemented is healthcare, in the Georgia Tech and Children's Healthcare of Atlanta MedVault project [6]. Figure 9 shows the architecture of our prototype service that maintains a personal health record (PHR) database by aggregating a patient's medical records from various sources. The patient has a personalized agent to provide authorization service for the PHR repository. The authorization scheme is policy based, where the patient can set his disclosure poli-

Input Type	Produce Graph	Verify Graph	Produce Chain		Verify Chain	
			Average	Median	Average	Median
Short strings	13,000,000	16,000,000	178	81	24,000	17,000
Full data	280,000,000	280,000,000	209	83	240,000	180,000

Table 2: Timings for the Ubuntu Package Graph (in μs)

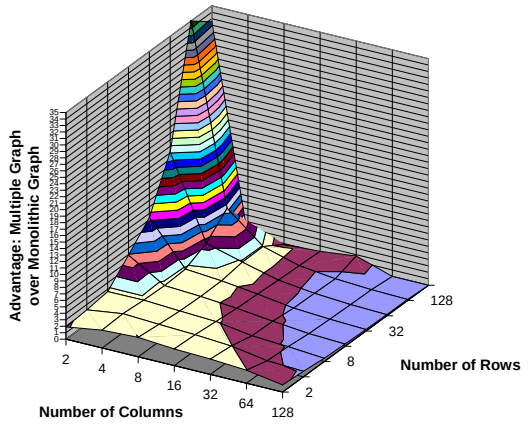


Figure 8: Relative Time to Verify a Single Dependency Chain

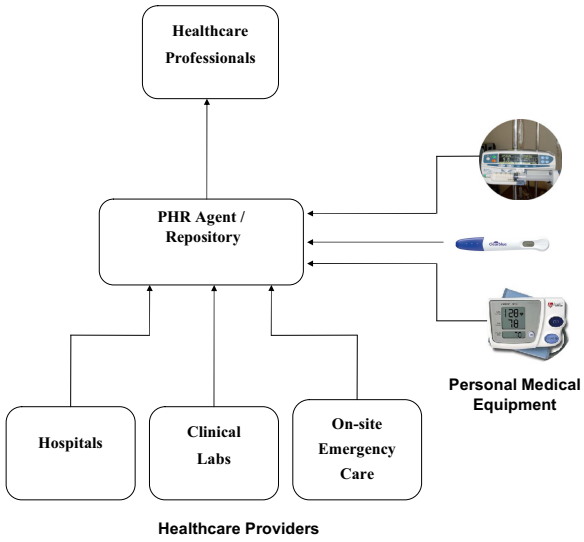


Figure 9: Architecture of the PHR service

cies. These policies are attribute-based and hence the patient defines a set of attributes that a requester must hold in order to access particular records. The repository holds the PHR which is composed of a number of medical records. The agent and the repository form a single unit as shown in the figure. This architecture is loosely based on the health care use case from [10], which can be considered a specification for systems that permit patients to access their own health records and to specify disclosure policies on those records.

Healthcare providers, such as hospitals, clinical labs, on site emergency care units etc., are entities that generate medical records about the patient and these records are added to the repository. These providers are the *certifiers* in our terminology and make use of the proposed dependency-aware redactable signature scheme in order to sign the records that they provide to the repository. The patient's personal medical devices also generate medical records about the patient and are useful in monitoring the patient's condition on a regular basis. The healthcare professionals require these personal health records to provide medical care to the patient. This medical care can be emergency, definitive or general. Healthcare professionals contact the patient's agent and request some records. Upon authentication and checking the authorization policies, the agent retrieves the records from the repository and sends them to the requesting professional. Although the medical records are being provided by the patient himself, the authenticity of the records can be verified as they are digitally signed by the originating entity.

The patient can set attribute-based authorization policies in his personalized agent. He specifies verifiable attributes that the requesting medical professional should possess in order to access documents from a particular category. We have implemented a policy engine that checks provided attributes against a set of XACML policies to determine if a particular request can be approved. Before doing this check, the request is modified to include any entries in the record that must be disclosed, because other items that have been requested are dependent upon them. In case the modified request is approved, the agent accesses the requested pieces of the health record and the additional depended-upon entries, and sends them to the requester along with additional hash values needed to verify the integrity and authenticity of the disclosed entries.

At the time of writing this paper, a basic implementation of the architecture in Figure 9 has been completed and performance evaluation of the architecture has begun.

8. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation (under Grant CNS-CT-0716252), the Georgia Tech Health Systems Institute, 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.

9. REFERENCES

[1] D. Bauer, D.M. Blough, and D. Cash. Minimal information disclosure with efficiently verifiable

credentials. In *ACM CCS2008 DIM Workshop*, Oct 2008.

[2] D. Bauer and D.M. Blough. Analysis of a redactable signature scheme on data with dependencies. Technical report, CERCS, Georgia Institute of Technology, 2009.

[3] Google. Google Health, 2009.

[4] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 89–98, New York, NY, USA, 2006. ACM.

[5] R. Johnson, D. Molnar, D. Song, and D. Wagner. Homomorphic signature schemes. In *Topics in Cryptology – CTRSA 2002*, vol. 2271, pp. 244–262. Springer-Verlag, 2002.

[6] Medvault project web site. `http://medvault.gtisc.gatech.edu/`.

[7] R. Merkle. A certified digital signature. In *Advances in Cryptography*, pp. 218–238. Springer-Verlag, 1989.

[8] Microsoft. Microsoft Healthvault, 2009.

[9] National Institute of Standards and Technology. Fips 180-2, secure hash standard, federal information processing standard (FIPS), publication 180-2. Technical report, Department of Commerce, August 2002.

[10] Office of the National Coordinator for Health Information Technology. Consumer empowerment: Consumer access to clinical information, June 2007.

[11] K. E. Seamons, M. Winslett, and T. Yu. Limiting the disclosure of access control policies during automated trust negotiation. In *Network and Distributed System Security Symp. (NDSS 2001)*. IEEE Computer Society Press, February 2001.

[12] A. Squicciarini, E. Bertino, E. Ferrari, F. Paci, and B. Thuraisingham. PP-trust-X: A system for privacy preserving trust negotiations. *ACM Transactions on Information and System Security*, 10(12), 2007.

[13] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[14] D. Wagner. A generalized birthday problem. In *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, pp. 288–303. Springer-Verlag, 2002.

[15] W. H. Winsborough and N. Li. Safety in automated trust negotiation. *ACM Trans. Inf. Syst. Secur.*, 9(3):352–390, 2006.

[16] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Negotiating disclosure of sensitive credentials. In *2nd Conference on Security in Communication Networks*, 1999.