On the Performance, Feasibility, and Use of Forward-Secure Signatures

Eric Cronin
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
ecronin@eecs.umich.edu

Sugih Jamin*
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
jamin@eecs.umich.edu

Tal Malkin†
Department of Computer Science
Columbia University
New York, NY 10027
tal@cs.columbia.edu

Patrick McDaniel
AT&T Labs–Research
Florham Park, NJ 07932
pdmcdan@research.att.com

ABSTRACT

Forward-secure signatures (FSSs) have recently received much attention from the cryptographic theory community as a potentially realistic way to mitigate many of the difficulties digital signatures face with key exposure. However, no previous works have explored the practical performance of these proposed constructions in real-world applications, nor have they compared FSS to traditional, non-forward-secure, signatures in a non-asymptotic way.

We present an empirical evaluation of several FSS schemes that looks at the relative performance among different types of FSS as well as between FSS and traditional signatures. Our study provides the following contributions: first, a new methodology for comparing the performance of signature schemes, and second, a thorough examination of the practical performance of FSS. We show that for many cases the best FSS scheme has essentially identical performance to traditional schemes, and even in the worst case is only 2-4 times slower. On the other hand, we also show that if the wrong FSS configuration is used, the performance can be orders of magnitude slower. Our methodology provides a way to prevent such misconfigurations, and we examine common applications of digital signatures using it. In addition to the evaluation methodology and empirical study, a third contribution of this paper is the open-source FSS library developed for the study.

We conclude that not only are forward-secure signatures a useful theoretical construct as previous works have shown, but they are also, when used correctly, a very practical solution to some of the problems associated with key exposure in real-world applications. Through our metrics and our reference implementation we provide the tools necessary for developers to efficiently use forward-secure signatures.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Public key cryptosystems

General Terms

Performance, Design, Security

Keywords

forward-secure signatures, digital signatures

1. INTRODUCTION

The Key-Exposure Problem

Digital signatures play an essential role for security on the Internet. Electronic commerce, private and authenticated communication, and secure storage are but a few of the multitude of applications that rely on signatures to assert authenticity, ownership, or delegation. However, signature-based systems are very vulnerable to the *key exposure problem*, which in practice is a far more likely cause of compromise than cryptanalysis. Once a private key has been exposed, not only are all future signatures associated with the compromised key suspect, but all *past* signatures as well, since there is no secure way to tell that a signature was generated before or after the compromise. All previous signatures must thus be indirectly revoked. The damage of these compromises can be enormous, both in terms of overhead in revoking and reissuing all past signatures,

*Sugih Jamin is supported in part by the NSF CAREER Award ANI-9734145, the Presidential Early Career Award for Scientists and Engineers (PECASE) 1998, and the Alfred P. Sloan Foundation Research Fellowship 2001. Additional funding is provided by AT&T Research, and by equipment grants from Sun Microsystems Inc., Compaq Corp., and Apple Inc. Part of this research was done when Sugih Jamin was at the University of Cambridge and the University of Tokyo.

†Part of this research was done while Tal Malkin was at AT&T Labs–Research.

CCS'03, October 27–30, 2003, Washington, DC, USA.

and in terms of the new security vulnerabilities introduced by the possibility of repudiation.

For example, the compromise of a Certificate Authority's (CA's) root private key results in *all* certificates from that CA being unverifiable until clients are updated with the new root public key, not just new certificates signed by the replacement key. All existing certificates (signed public keys) must also be revoked and re-certified with the new key, as there is no way to verify that they were signed prior to the compromise. Where the root CA is popular (e.g,. Verisign), the compromise could lead to widespread disruption of the Internet. The digital signing of legal contracts illustrates another environment where the possibility of key exposure severely weakens security. By purposely exposing their private key, a party that wishes to back out of a contract gains the ability to repudiate their previous signature at any time by claiming it to be forged. As another example, consider an electronic checkbook application, in which each check is created by the account holder signing an amount, a recipient, and the current date using the checkbook's private key. The recipient submits the electronic check and the signature to the account holder's bank, and the check is honored if the signature is deemed valid. Here, if the private key is exposed, it is incumbent upon the bank and account holder to investigate and possibly repudiate every check received by the bank, possibly even valid uncashed checks.

The above examples illustrate that limiting the effect of key exposure should be afforded at least as much effort as prevention of cryptanalysis. However, due to lack of useful countermeasures, this design goal has not been widely embraced in existing systems. Currently, the most widely used techniques to limit damage due to key exposure are the use of short lived keys and centralized timestamping services. Neither of these solutions scales well: the former requires many public keys to be certified, distributed, and maintained, while the latter requires a trusted third party who itself must never be compromised in order for security to be maintained.

Forward-Secure Signatures and Applications

Forward-secure signatures (FSS), first proposed by Anderson [2] and formalized by Bellare and Miner [5], have recently emerged as a promising viable mitigation technique for key exposure. FSS differ from traditional signatures in that the private keys are periodically updated via a one-way process that produces a new private key corresponding to the same public key. The FSS construction guarantees that once updated, past iterations of a private key cannot be recovered from the new private key, implying that the exposure of the current private key does not render past signatures suspect. The key exposure problem is therefore mitigated by FSS schemes, since any material that could be used to forge a past signature is destroyed.[1] The next section provides background into the different types of existing forward-secure signatures, particularly those used in our empirical study.

FSS can potentially change the semantics of certificates and signatures in a fundamental way, as we illustrate by revisiting some of the examples above. Using a FSS-enabled CA, certificates are signed using the (possibly one-time) private key, and only released after the private key is updated. Hence, subsequent compromise of the CA root key only affects those certificates issued at or after the point of compromise. The amount of disruption is bounded by the number of potentially compromised certificates, not by the number of certificates issued under the compromised key. For the electronic checkbook application, implemented with an FSS scheme, in the event of a compromise the account holder tells the bank which version of the private key was lost. Every prior signature could be processed as normal, and signatures associated with current and future private keys systematically voided. In this case compromise prevents future use of the checkbook, but does not put past checks under a cloud of suspicion.

Many other applications can potentially be enabled by FSS semantics. For example, an application can limit the monetary amount of data signed by a particular key. Once the cumulative value of signed transactions meets a threshold, the singing key is updated. This can be used as a kind of one-time credit card (e.g., phone cards). Betting slips, electronic receipts, airline tickets, legal archives, and time-stamping services would also benefit from the use of FSS services.

One can view FSS as check-pointing a stream of signatures. Whether a key is updated after every signature, once a day, or under any other policy is a trade-off between performance (e.g., cost of the update operation) and security (e.g., vulnerability to forgery). Once an update has occurred, it is infeasible for previously checkpointed signatures to later be forged. Assuming she conforms to the algorithm, even the signer cannot forge past signatures. This latter property may enable novel applications. For example, signer forgery prevention mechanisms would be of great value in the retention of legally binding documents (e.g., official billing records).

Our Contributions

Forward-secure signatures are a strictly stronger cryptographic construction than traditional signatures: they provide all the properties of traditional signatures as well as the new forward-security property to protect against key exposure. Due to this, it is expected that once implemented there will be some (possibly significant) decrease in performance for FSS compared to traditional signatures. Previous studies of forward-secure signatures have evaluated the design and asymptotic performance of these cryptographic constructions, but have not considered their practical application. This paper develops an understanding of the real world performance characteristics through a comprehensive empirical study of forward-secure signatures.

One of the challenges of performing this type of study is the lack of systematic metrics for evaluating the inherent tradeoffs of signature systems. Instead of simply looking at individual operation costs in our study, as one of the contributions of this work we develop an evaluation metric in which operation costs are amortized based on expected use over the lifetime of the key(s). We believe this new usage-based metric is more robust because it encompasses the application and environment, not just cycle consumption of basic operations.

Using this metric we perform an empirical study of the performance of several forward-secure signature schemes. Our study reveals several notable results pertaining to the performance of these schemes, and of traditional non-forward-secure signature schemes as well. We begin with an example result that highlights the ability of our methodology to capture many different applications and environments in its analysis. When using traditional signature schemes, there is a somewhat common belief that, due to very efficient verification, RSA [36] is the best algorithm to use in all situations. Our analysis dispels this belief, showing that while for many applications this is true, and RSA is up to fourteen times faster than other algorithms, it is not always true. For other environments, which we show later represent real-world applications, RSA actually performs up to twenty-four times *slower* than another signature algorithm, ECDSA [4]. The metric allows us to not only see which

[1] This assumes that the update operation is actually able to permanently destroy *all* copies of the old version of the key. This assumption is provably necessary for any FSS method. Section *4.2.2* examines the practical impact of this requirement.

algorithm is most efficient at a particular point, but also to visualize performance over the entire space of environments.

The primary result of our study, and a major contribution of this paper, is an accurate view of the performance characteristics of different forward-secure signature constructions and parameters for different applications. We show that no single construction performs well in all situations, and that knowing the performance characteristics of different schemes is critical in avoiding potentially enormous overheads. By using the optimal forward-secure scheme, our results show that for many applications the overhead of FSS over the optimal traditional signature scheme is actually almost zero, and even in the worst case, the difference between optimal forward-secure and traditional schemes is only between a factor of two and four. We also show that choosing the correct forward-secure configuration has significant risks, and the optimal configuration for one application can be several orders of magnitude worse than other FSS schemes for another application.

Finally, another contribution of this work is the reference implementation of forward-secure signature schemes used in our study. This library uses the common OpenSSL [38] cryptographic library and provides an API closely based on OpenSSL's own API for non-forward-secure signatures. It utilizes all base signature schemes provided by OpenSSL and currently contains five forward-secure schemes. This library will be released under an open-source license and available for download by anyone interested in incorporating forward-secure signatures in their applications.

An outline for the remainder of the paper follows. In the next section we overview of FSS schemes and their theoretical (asymptotic) performance characteristics. We then present our empirical study in the following section, exploring the actual performance of these schemes. Section 4 describes the reference implementation of FSS used in our study, as well as several unexpected issues which arose during its development. Finally, we conclude and summarize our findings and present ideas on future work.

2. FORWARD-SECURE SIGNATURES

In this section we survey known forward-secure signature methods, together with their salient security and efficiency properties, as theoretically analyzed by the authors of each method. We particularly focus on the schemes that will be evaluated in this work, which include most of the best methods known to date. Finally, we touch upon other related work.

2.1 Some Simple Solutions

Consider the following trivial forward-secure signature scheme, with a parameter T denoting the total number of time periods over which the scheme is supposed to operate. Starting from any standard signature scheme as a base, the signer runs the key generation protocol T times, obtaining T secret/public key pairs $(sk_1, pk_1), \ldots, (sk_T, pk_T)$. The public key is now set to $PK = (pk_1, \ldots, pk_T)$, while the secret key (for the first time period) consists of $SK_1 = (sk_1, \ldots, sk_T)$. For each time period $1 \leq j \leq T$, signing and verifying are performed using the base scheme relative to the secret key sk_j (for signing) and public key pk_j (for verifying). To update from time period j to $j+1$, the signer simply erases the key sk_j (which is no longer necessary).

Clearly, this scheme is secure as long as the underlying standard signature scheme is. Signature and verification time, as well as signature size, are very efficient in this scheme – the same as in the underlying base scheme. Still, this trivial scheme is clearly not practical, because it requires public and secret key sizes which are linear in T, the number of time periods the schemes can be used for.

Some improvements to the above scheme were suggested already in the first works in the area [2, 5, 27]. For example, Krawczyk [27] proposes a method where the total storage of the signer is still linear in T, but the public key and the secret key themselves are both short (proportional to the security parameter). That is, the bulk of the keying material stored is needed by the signer, but even if compromised by an adversary, forgery is still not possible.

While these improvements are better than the trivial scheme above, for a practical scheme it is desirable not to have such a linear dependence on T, in any of the parameters. We further discuss the efficiency requirements below.

2.2 Efficiency Requirements

What are the efficiency requirements when designing a forward-secure signature scheme? We cannot hope to beat standard signature schemes, since forward-secure signature schemes are a strictly stronger construct. Thus, we would like a forward-secure signature scheme to perform not much worse than a standard one, in terms of time (for key generation, signing and verifying), and space (key size and signature size). In particular, it is not desirable for these parameters to grow with the number of time periods T.

One might be tempted to require that there is no dependency whatsoever on T. But note that, as pointed out by [28], signature schemes already depend on some security parameters, which must be super-logarithmic in T. If this were not the case, by the time period T is reached, the scheme could have been broken by exhaustive search. Therefore, a logarithmic asymptotic dependence on T is no worse (and possibly better) than the required linear dependence on the security parameters.

Indeed, to evaluate the performance of a forward-secure signature scheme, it is necessary to examine the actual performance more closely. The only type of analysis of forward-secure schemes to date involves a theoretical analysis, estimating the asymptotic performance of the schemes as a function of T and two security parameters (representing the key-sizes for private-key operations such as hashing, and public-key operations such as RSA signatures). In this work we suggest the first experimental performance evaluation, for the forward-secure signatures described below.

We next turn to reviewing the main features of known forward-secure signature schemes. These can be divided to two main categories, as detailed below.

2.3 Generic Provably-Secure Schemes

This category includes schemes that can use any arbitrary base signature scheme, and their security is provable as long as the base signature scheme is secure. For example, the trivial solutions discussed above fall under this category. Efficient generic provably secure schemes (without a linear dependence on T) are the ones implemented in this work, and surveyed below (we refer the reader to the original papers for more details on the schemes).

Bellare-Miner Tree

Bellare and Miner [5] suggest a forward-secure scheme based on a binary certification tree. Roughly speaking, a binary tree is constructed with T leaves, where each leaf corresponds to a time period. An instance (public-key,secret-key pair) of a base (standard) signature scheme is associated with each node in the tree. The public key of the forward-secure scheme is the public key of the root, and the secret key consists of all the base key pairs associated with nodes on the path from the root to the current time period. A signature for time period j consists of a certification chain from the appropriate leaf to root, where the leaf key is used to sign the actual message, and each node is used to sign the public key of its child.

Verification proceeds by verifying each signature on the chain, up to the top level (root) signature which is verified against the public key.

In this scheme, all performance parameters (size and time) are $poly(k,l)O(\log T)$, where k, l are the security parameters (the actual dependence on k, l depends on the performance of the underlying base scheme). As explained above, this logarithmic dependence on T is very good, and, at least from a theoretical point of view, this scheme is satisfactory. Still, this is not competitive with standard signature schemes, as signing a message involves $\log T$ signatures (which are expensive public-key operations) in the underlying base scheme. Allowing a high number of possible time periods T, this factor can be prohibitive. On the other hand, key update is very efficient, roughly consisting of two (amortized) key generations in the underlying base scheme (since each key in the tree is only generated once).

Product Construction

Malkin, Micciancio, and Miner [28] suggest two composition operations, taking any two forward-secure schemes with T_1, T_2 time periods, respectively,[2] and constructing a new forward-secure signature scheme with more time periods. These constructions are suggested as tools in constructing flexible forward-secure scheme by applying them repeatedly in different combinations, possibly with other known schemes. The constructions are also used toward the main [28] scheme.

Their first composition operation is the product construction, resulting in a scheme with $T_1 \cdot T_2$ periods. Here, an instance of the first scheme (with T_1 periods) is generated, and for each time period, a new instance of the second scheme (with T_2 periods) is generated underneath it. The public key is the public key of the top (first) scheme, and a signature consists of signing the actual message with the key of the second scheme (in the appropriate time period), and signing the public key of the second scheme with the key of the first scheme.

The product construction can be viewed as making explicit the main building block that was already used by the Bellare-Miner tree (as well as Anderson's original scheme [2], and other certification based constructions). Indeed, when iterated recursively, starting from any standard base scheme, the product construction results in a scheme which is essentially the same as the Bellare-Miner tree. In this work we evaluate both the Bellare-Miner tree (iterative product construction), and an application of the product construction on two schemes resulting from iterative sum construction, described below.

Sum and Iterated Sum Constructions

The second composition operation suggested in [28] is the sum construction, combining two schemes with T_1, T_2 time periods, respectively, to a scheme with $T_1 + T_2$ time periods. Here, an instance of each of the two schemes is generated, and the public key is the hash of both public keys. A signature consists of the two public keys, and a signature of the message according to the first (if it is within the first T_1 time periods), or the second (if it is within the next T_2 periods). Taking the secret key to consist of the (secret and public) keys of both schemes would allow the right functionality for the signer, but result in an inefficient key size. Instead, a more efficient way to generate and maintain the secret keys is proposed, and the reader is referred to [28] for details.

When iterated recursively, starting from any standard base scheme, the iterated sum construction results in a binary tree, with each time period associated to a leaf, similar to the Bellare-Miner tree. However, each node in the tree is the hash of its two children (in the same spirit of Merkle trees [30]), rather than being used to sign its children. This means that signing (and verifying) is much more efficient, as $\log T$ hashes (private key operations) are extremely fast to compute, compared with $\log T$ signatures, which are slow. On the other hand, key update is much slower here, since the signer must generate all the public keys in a bottom-up fashion, in order to compute their hash which is the public key; since the signer does not keep all keys (or else the key size would be very large), regeneration of keys is necessary during the life time of the scheme, resulting in (amortized) $\log T$ key generations per update. In contrast, in the Bellare-Miner scheme, the tree is built top down, with new keys generated only when needed, resulting in 2 key generations per update.

In sum, the asymptotic performance of the iterated sum construction is also $poly(k,l)\log T$; when explored more carefully, signing and verifying is extremely efficient, consisting of one signature in the base scheme plus $\log T$ hashes, while key generation and updates take about $\log T$ key generations of the base scheme, which may be slow ([28] suggests that this may be improved using one-time signatures[14, 8]).

MMM Tree

The main scheme suggested in [28] is one using both the sum and product constructions, starting from any underlying base signature schemes. The idea is to use an iterated sum construction with a polynomial number of time periods, where for each time period, another iterated sum construction is attached, using the product construction. The number of periods for each iterated sum construction on the lower level keeps increasing as time progresses, starting from a 2-period scheme attached to the first time period of the top level, then a 4-period scheme attached to the second period, an 8-period scheme for the third period, and so on.

The MMM construction achieves a new feature that previous schemes did not, namely that the maximal number of time periods, T, need not be fixed in advance, and thus does not influence the performance. Rather, more time periods are added as needed, for an arbitrarily large polynomial number of time periods. The performance slowly degrades, proportional to $\log t$, where t is the number of time periods elapsed so far.

In terms of performance, signing consists of two signatures plus $\log t$ hashes in the underlying base scheme (which is very efficient), while key update consists of $\log T$ key generations in the base scheme (which may be expensive). The main performance advantage of [28], namely that of efficient signing and verifying, is due to the fact that the iterated sum construction is prominently used. The use of product construction may in principle make MMM a little less efficient than iterated sum, but it allows for unbounded number of time periods, which also makes the performance better, as it only depends on the number of time periods elapsed so far, and not on the maximum T. This is what the theoretical analysis of the schemes indicates. An experimental performance analysis is conducted in this paper, and compares MMM, iterated sum, product, and Bellare-Miner tree in different settings.

2.4 Specific, Random Oracle Based Schemes

Another category of forward-secure schemes consists of constructions built upon *specific* standard signature schemes, which in turn (in all the known schemes) are built from specific identification protocols using the Fiat-Shamir methodology [18]. These FSS schemes (as well as the underlying standard signatures) are proven

[2] Note that a standard signature scheme can be viewed as a forward-secure scheme with one time period.

secure in the random-oracle model.

The first proposed forward-secure signatures in this category are the main scheme of Bellare and Miner [5] (based on the Ong-Schnorr [34] scheme), optimizing key and signature sizes, followed by a scheme of Abdalla and Reyzin [1], which improves the time parameters at the expense of longer key and signature sizes. However, these schemes still have signing and verification times which are linear in T.

Itkis and Reyzin [22] propose a FSS (based on the Guillou-Quisquater [19] scheme), which does not have linear dependence on T in any of its parameters, and optimized signing and verifying time. In particular, signing and verifying in their FSS scheme is comparable to (about twice as expensive as) the underlying [19] standard signature scheme.

Kozlov and Reyzin [26] propose a FSS with a very fast update operation, which significantly improves upon all other forward-secure signatures (generic or specific). However, the performance of other parameters (such as signing and verifying time) are not as good for this scheme, and so it is only useful for specific applications where update time is the important parameter to optimize.

2.5 Comparison

The main advantage of the generic schemes is that they have provable security (assuming any signature schemes exist[3]). This is in contrast to the specific schemes which can only be proven secure in the random-oracle model, a weakness they inherit from the underlying standard signatures they are based on. Proven security in the random-oracle model is very valuable as a heuristic for security, but provides a significantly weaker security guarantee. In particular, it does not even guarantee the existence of an instantiation of the random oracle for which the scheme is secure [9].

Another advantage of generic schemes stems from the fact that they can be used with any underlying base signature schemes. This again provides a stronger security guarantee (as it requires a weaker assumption – any implementation of one-way functions suffices). Furthermore, this allows for flexibility in optimizing and trading off the time and space parameters of the FSS scheme, by using base schemes with different performance characteristics, rather than being bound to the properties of a specific base scheme. Finally, the generic MMM scheme has the advantage of an unbounded number of time periods, as discussed above.

The main advantage of the specific schemes, is that they achieve better dependence on T and the security parameters in some of their time or space measures. For example, the main scheme of Bellare and Miner [5] achieves key and signature sizes which are completely independent of T. Moreover, the performance of each of these specific schemes is typically significantly better, at least in some of the parameters, than the best generic schemes that were known at the time the specific scheme was introduced. In particular, Itkis and Reyzin [22] were the first to propose a FSS scheme with signing and verifying which is comparable to standard signature schemes, while maintaining reasonably good performance (in particular, not linearly dependent on T) in all other time and space parameters. (The MMM generic scheme described above, which also achieves this, was proposed later.)

Finally, specific schemes, and in particular that of [22], were the basis for several extensions to other models [17, 23], as well as the fast update FSS scheme of [26].

Comparing a generic scheme with a specific scheme in terms of asymptotic performance (as done by the authors of the various schemes), is difficult. This is because the different instantiations of the generic scheme give different results, typically trading off improvements in different parameters. For a most informative general comparison, it is probably best to instantiate the generic scheme with the same base signature scheme as the specific scheme under consideration. For a comparison geared toward a particular application, the generic scheme should be instantiated with the best possible base scheme for that application.

With this in mind, the asymptotic performance of the best (considering all parameters) specific scheme [22] and the best generic scheme [28] are quite close (and neither is better than the other in all parameters simultaneously). For both, the theoretical analysis predicts that they should be feasible.

2.6 Schemes Implemented in This Work

In this paper we focus on the generic Bellare-Miner tree, iterated sum, product, and MMM constructions. In several places we refer to these schemes by short names, BMTree, ISum, Prod, and MMM respectively. We chose the generic schemes due to the advantages outlined above. Moreover, the modular nature of these tree based constructions, together with the possibility of using different standard signatures as a base, provides much flexibility in constructing the final scheme. It is thus important to evaluate performance of particular choices in order to determine the feasibility and applicability of forward-secure signatures for desired applications. We note that it may also be interesting to evaluate performance of combined specific and generic schemes, e.g., applying the sum construction on top of the Itkis-Reyzin construction from [22] for a smaller size.

2.7 Other Related Cryptographic Primitives

Forward-security was first considered for key-exchange protocols [20, 15]. Forward-security for private-key primitives (pseudo-random generators, authentication, and encryption) was proposed in [6], and for public-key encryption in [10].

Several extended models for forward-secure signatures have been proposed. These include forward-secure group signature schemes [37], key insulated signatures [16, 17], and intrusion resilient signature schemes [23], combining the benefits of forward-secure and key-insulated signatures. In particular, intrusion resilient schemes have the advantage that if the key is exposed for some number of time periods, signatures in all other time periods (not just the past ones) are still valid. This is achieved by distributing the signer to two modules, one for signing, and one is a base which is needed every time an update is performed. Security for all compromised time periods is maintained as long as the base and signer modules are not both compromised in the same time period (if they are, then only forward-security is guaranteed). Studying the feasibility of this distributed model is an interesting problem for future work.

3. PERFORMANCE EVALUATION

Digital signature performance, whether it be for traditional or forward-secure signatures, is all about tradeoffs. Balancing these tradeoffs in order to select an optimal set of parameters is not always a straightforward task. In this section, we present several metrics for formalizing these tradeoffs between different aspects of signature performance for both traditional and forward-secure signature schemes. To our knowledge, no previous work has utilized such a technique capable of evaluating signature schemes' performance over the entire spectrum of uses. We then evaluate the performance of signature schemes with regard to these metrics using benchmarks of several of the FSS schemes introduced in the previous section. We show that when configured correctly, the

[3]Equivalently, assuming one-way functions exist.

overhead of FSS compared to traditional signature schemes can be quite small: in the best case, the performance of FSS is identical to that of non-forward-secure schemes; in the worst case, FSS is only three and a half times slower, and on average it is less than twice as slow. We also show that when configured poorly, FSS can be enormously expensive compared to traditional signatures, emphasizing the need to evaluate the intended use before selecting a configuration. We look at where specific uses of digital signatures lie in the space defined by our metrics, and which FSS configurations are optimal for these situations.

3.1 Performance Metrics

Key generation (or update) time, signature time, and verification time are all indicators of a signature scheme's performance. However, no one aspect alone is enough to judge whether one signature scheme is better than another for all situations. Many earlier performance comparisons take an informal approach at resolving this problem by first looking at a specific situation and then picking which operation seems to be most important for it [40]. This works well for simple situations but does not help when it is unclear which operation is most important or in seeing the entire picture with regards to performance tradeoffs.

Instead of taking a similar approach for our analysis, we look at what characteristics make a given environment using signatures unique, and how to express this as a set of parameters. We define several metrics using these parameters which compute a single amortized cost for the performance, allowing us to make direct comparisons between schemes for any given situation. Using this evaluation framework, we are able to not only look at specific cases as previous performance studies have done, but also at how the performance changes over the entire range of parameters, and how different signature schemes perform in this broader picture.

3.1.1 Traditional Signature Schemes

When examining common uses of digital signatures, we see that what makes each situation unique is the frequency at which each of the three basic operations must be performed. A Certificate Authority (CA), for example, will generate a single key. It will then use that key to produce hundreds or thousands of signatures on certificates, and each of those signatures may be verified thousands of times as the certificates are used. An electronic checkbook system, in contrast, will require a key generation for every checkbook issued. Each checkbook (key) can then produce hundreds of checks (signatures), each of which will likely only be verified once, when "cashed" at the issuing bank. Below, we express these relationships between key generation, signature, and verification rates (G, S, and V respectively) through two ratios, $\mathcal{R}_1$ and $\mathcal{R}_2$. These two ratios allow us to combine the individual costs of each operation (C_g, C_s, and C_v respectively) proportionally to arrive at a single cost for a given scheme given a particular mix of key generations, signatures and verifications.

We begin by examining the tradeoff in the cost incurred by the signer versus the cost incurred by the verifier(s). One way to express both the signer's and verifier's costs in a single value is to "tax" each verification for its share of the work performed by the signer in producing the signature. Looking at the scenario of a CA as described in the previous paragraph, we see that including a share of the signing cost adds very little to the cost of any one verification since there are so many verifications. On the other hand, another common use for signatures is during the session negotiation of protocols such as SSL [13]. Here, a signature is produced, verified once, and then forgotten. The entire cost of the signature's generation is now associated with that single verification as opposed to thousands of verifications as before, and its cost is far more important to the overall performance. In the middle of these two extremes lie usages such as digitally signing contracts. Here, the cost of signing may be spread out over a few verifications, reducing its impact but not making it possible to ignore it altogether.

$\mathcal{R}_1$ and the metric $\mathcal{M}_1$ express this relationship formally. $\mathcal{R}_1$ is the ratio between signatures and verifications for the workload in question. $\mathcal{M}_1$ produces a weighted verification cost that incorporates an equal share of the cost of signing into each verification.

$$\mathcal{R}_1 = \frac{V}{S} \quad (1)$$

$$\mathcal{M}_1 = C_v + \frac{1}{\mathcal{R}_1} \cdot C_s \quad (2)$$

On the signer's side, a similar tradeoff exists between the "offline" cost of key generation, and the "online" cost of actually generating signatures. Here, it is useful to look at amortizing the shared key generation costs over each signature produced. A busy SSL server may receive millions of connections over the lifetime of its private key. If each of these secure connections requires a signature to be produced, then the share of key generation that should be attributed to each signature approaches zero. On the opposite end of the spectrum, if the use of one-time signatures [14, 8] is required, then by definition each key may only be used to produce a single signature and then must be destroyed. The second ratio and metric $\mathcal{R}_2$ and $\mathcal{M}_2$ expresses this relationship.

$$\mathcal{R}_2 = \frac{S}{G} \quad (3)$$

$$\mathcal{M}_2 = C_s + \frac{1}{\mathcal{R}_2} \cdot C_g \quad (4)$$

Neither of these metrics alone is sufficient, however. The cost of a signature used in $\mathcal{M}_1$ must take into account the amortized key generation costs from $\mathcal{M}_2$ in order to accurately reflect the total cost. We define one final metric, $\mathcal{M}_3$, which combines the previous two metrics to give a weighted cost of verification incorporating both the balance between verifier and signer and the balance between the signer's "online" (signing) and "offline" (key generation) costs. This metric is parameterized by both $\mathcal{R}_1$ and $\mathcal{R}_1$, and gives us a single value with which to compare signature schemes.

$$\mathcal{M}_3 = C_v + \frac{1}{\mathcal{R}_1} \cdot C_s + \frac{1}{\mathcal{R}_1 \mathcal{R}_2} \cdot C_g \quad (5)$$

3.1.2 Forward-Secure Signature Schemes

Turning now to forward-secure signature schemes, a fourth operation, key update, is added to the the three operations present in traditional schemes. Like key generation, update is an "offline" cost, necessary before a signature can be performed but not actually part of the signing process. Key generation can be viewed as a special case of update, going from a null private key to the initial private key for period 0. Therefore, we can combine key generation and update into a single parameter U (with cost C_u), and use this in place of G and C_g in our metrics. $\mathcal{M}_1$, which was not influenced by G, remains the same. The modified $\mathcal{R}_2$, $\mathcal{M}_2$, $\mathcal{R}_3$, and $\mathcal{M}_3$ we denote with a star:

Short-Term Security			
	RSA 1024	DSA 1024	ECDSA t163k1
keygen (C_g)	352	5,500	**3.68**
sign (C_s)	10.5	4.36	**3.75**
verify (C_v)	**0.540**	5.35	7.61
Long-Term Security			
	RSA 1536	DSA 1536	ECDSA t223k1
keygen (C_g)	862	58,500	**6.72**
sign (C_s)	27.6	8.78	**6.81**
verify (C_v)	**1.03**	11.1	13.6

Table 1: Base signature scheme performance, in milliseconds.

$$\mathcal{R}_2^* = \frac{S}{U} \quad (6)$$

$$\mathcal{M}_2^* = C_s + \frac{1}{\mathcal{R}_2^*} \cdot C_u \quad (7)$$

$$\mathcal{M}_3^* = C_v + \frac{1}{\mathcal{R}_1} \cdot C_s + \frac{1}{\mathcal{R}_1 \mathcal{R}_2^*} \cdot C_u \quad (8)$$

The number of signatures per key generation ($\mathcal{R}_2$) still has importance with forward-secure schemes, but at a higher level than what these metrics are designed to measure. For FSS (as well as traditional) signature schemes, $\mathcal{R}_2$ also expresses the administrative overhead of the key for certifying and distributing the public key. As this cost is highly situation dependent we do not attempt to quantify it, but simply note that a higher value of $\mathcal{R}_2$ indicates that the cost of making the public key known and trusted is able to be amortized over more signatures.

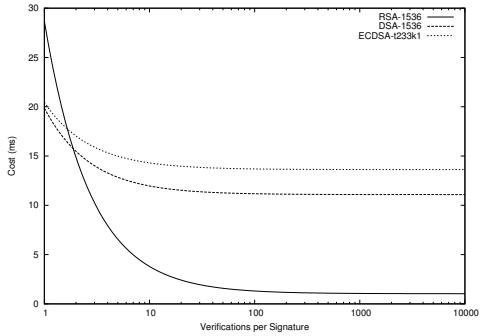
3.2 Experimental Setup

Having constructed a framework for comparing the performance of the signature schemes implemented in this paper, the next step in our evaluation is to fill in the needed values for C_g, C_u, C_s, and C_v. We use a simple micro-benchmark built with `libfss` called `fssbench` to measure these costs. For each configuration, the benchmark application times the initial generation of a combined public/private key followed by the three periodic operations during each period of the private key's lifetime.[4]

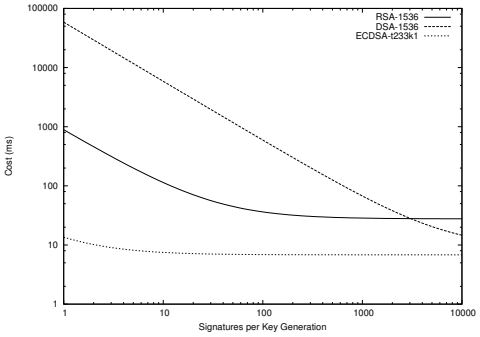
This is repeated for each configuration a total of ten times, with a different initial random seed used in key generation for each iteration. The results over all ten runs are averaged, and the results for multi-period operations are averaged again over the range of periods examined in order to provide a single average cost for each operation. For example, when comparing the performance for a situation requiring a maximum period of 512, only the first 512 updates of a 65535 period key would be included in the averages. Because of this, for keys with very large maximum periods we are able to stop benchmarking once this maximum examined period is reached.

We benchmark four of the generic FSS constructions described in Section 2: Bellare-Miner Tree (BMTree), Iterated Sum (ISum), Product (Prod), and MMM. In addition, we consider three base algorithms: RSA, DSA [31], and Elliptic Curve DSA (ECDSA).

[4] `fssbench` actually measures a great deal more, as described in [3]. For conciseness, we limit our discussion to those features of `fssbench` used in this study.



(a) $\mathcal{M}_1$.



(b) $\mathcal{M}_2$.

Figure 1: Performance of long-term traditional signature schemes.

The benchmarks were performed on a 1.5GHz Pentium 4 with 1GB of memory running FreeBSD 4.8. A development snapshot of OpenSSL 0.9.8 from April, 2003 was used for the cryptographic support library.[5] Both `libfss` and OpenSSL were built with compiler optimizations enabled.

3.3 Microbenchmark Results

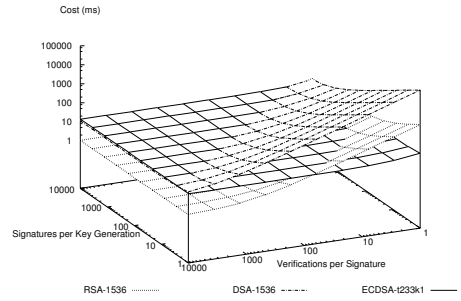
3.3.1 Base Algorithm Performance

We begin with an examination of the traditional signature schemes in order to determine which FSS configurations may be of particular interest to examine, as well as to introduce our metrics in a simplified environment. As mentioned, the OpenSSL cryptographic library used by `libfss` provides the three currently FIPS-approved signature schemes as potential bases: RSA, DSA, and ECDSA. Earlier forward-secure signature papers have mentioned several other signature algorithms of potential use for constructing forward-secure keys, including Guilleau-Quesquiter [19] and Fiat-Shamir [18]. We do not include these schemes in our comparison due to their limited use in existing real-world cryptosystems and lack of acceptance by standards organizations at the present time when compared to RSA, DSA, and ECDSA.

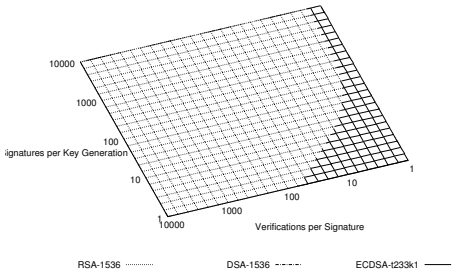
[5] Certain desired features of OpenSSL such as AES and full Elliptic Curve Cryptography support were not available in released versions of OpenSSL at the time of these experiments.

Short-Term Security								
	BMTree 256 RSA	ISum 256 RSA	Prod 16*16 RSA	MMM 255 RSA	BMTree 256 ECDSA	ISum 256 ECDSA	Prod 16*16 ECDSA	MMM 255 ECDSA
keygen	5,630	83,300	10,500	2,980	123	949	124	**40.8**
sign (C_s)	10.5	10.4	10.5	10.5	3.75	3.74	**3.73**	3.79
verify (C_v)	4.47	**0.576**	1.08	1.08	66.7	7.45	14.9	14.9
update	635	1,280	959	1,310	14.4	14.8	**11.5**	15.1
gen+up (C_u)	654	1,610	997	1,320	14.9	18.5	**12.0**	15.2

Table 2: Forward-secure signature scheme performance (first 255 periods), in milliseconds.



(a) $\mathcal{M}_3$.



(b) Lowest-cost scheme for $\mathcal{M}_3$.

Figure 2: Performance of long-term traditional signature schemes (cont.).

Table 1 summarizes the key generation, signing and verification times for the three base algorithms in 'short-term' and 'long-term' security strengths. The fastest result for each operation is in bold. Short-term security represents a keysize believed to be safe against brute-force attacks today, while long-term security represents a keysize believed to be safe for the next several decades [33]. There is no direct mapping between ECDSA group size and RSA or DSA key size, but a group of approximately the same strength as DSA or RSA given the current best known attacks on each scheme was chosen based on the recommendations in [11]; Koblitz curves were used due more efficient performance than other elliptic curves.

Using the costs in Table 1, we are able to plot the curves for $\mathcal{M}_1$ and $\mathcal{M}_2$ for each of the signature schemes. Fig. 1 shows $\mathcal{M}_1$ and $\mathcal{M}_2$ for the long-term keys. Both graphs have a logarithmic x-axis, and $\mathcal{M}_2$ has a logarithmic y-axis as well. The plots start with a 1 to 1 ratio on the x-axis at the origin. The results for short-term keys are very similar in behavior, and are omitted due to space constraints.

For $\mathcal{M}_1$, we see that there is a crossover point at two verifications per signature, after which RSA rapidly becomes much faster than either DSA or ECDSA. Meanwhile, ECDSA goes from nearly as fast as DSA initially to about 2.5ms slower, and both have a fairly constant cost after ten verifications per signature, by which point C_s contributes hardly anything to the cost.

Fig. 1(b) shows that for $\mathcal{M}_2$, ECDSA is faster than the other two schemes for all values of $\mathcal{R}_2$, and significantly faster until $\mathcal{R}_2$ grows very large. DSA starts out several orders of magnitude more expensive than either RSA or ECDSA when $\mathcal{R}_2$ is low. As the ratio approaches 1000 signatures per key generation, the cost stabilizes at C_s for DSA, which is only a few milliseconds slower than ECDSA. For RSA, the initial performance is still much worse than ECDSA's, but much better than DSA's. RSA converges on C_s much faster, and by 100 signatures per key generation C_g is contributing almost nothing to RSA's performance.

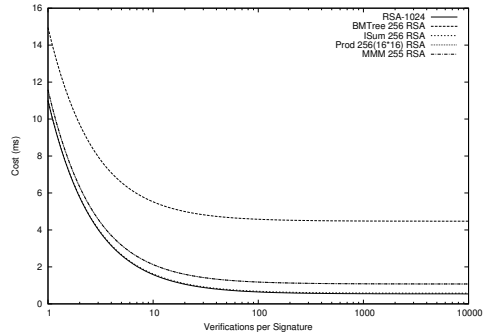
By combining these two metrics we get Fig. 2(a). This plot shows the three-dimensional surfaces defined by $\mathcal{M}_3$ for each of the schemes. Since we are concerned with the minimum of these surfaces, which is often obscured by the less efficient schemes, we also show the "view from below" in Fig. 2(b). This second graph must be interpreted with care, as it does not depict how much more efficient the lowest scheme may be compared to other schemes. In some cases, two schemes will be nearly tangent to one another when they intersect, in which case there is little difference in picking one scheme or the other until you move a significant distance from the intersection. Other times, as is the case in Fig. 2(b), the two surfaces intersect at a steeper angle, causing larger differences between schemes when moving away from the intersection.

From Fig. 2, we can clearly see the areas where particular schemes are most efficient. When both $\mathcal{R}_1$ and $\mathcal{R}_2$ are low, ECDSA performs the best due to significantly cheaper key generation. As the parameters increase, particularly the number of verifications per signature, RSA becomes much less expensive. As $\mathcal{R}_1$ and $\mathcal{R}_2$ increase, the amortized costs of key generation and signing approach zero, and the performance of each scheme converges on the cost of verification alone.

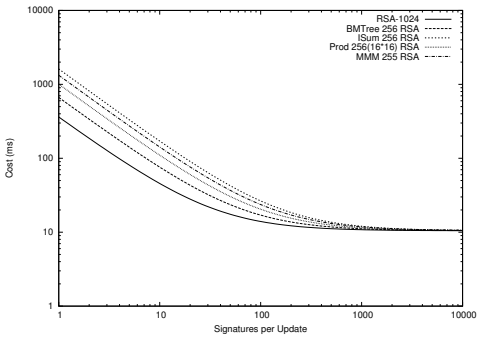
Looking at how this relates to our illustrative examples from Section 3.1, we see that the CA would fall solidly in the left portion of Fig. 2(b), where RSA is the dominant algorithm. For the SSL server, ECDSA's lower signing cost compared to RSA makes it a better choice. The case of digitally signing documents lies towards in the border region between RSA and ECDSA, with the popularity of the signer (i.e. how many signatures they produce over their key's lifetime) being the determinant for which scheme to chose. As more signatures are generated, RSA becomes the preferable algorithm.

3.3.2 Forward-Secure Signature Performance

Continuing our analysis, we examine forward-secure signature schemes in a similar manner. We assume for now that the maxi-

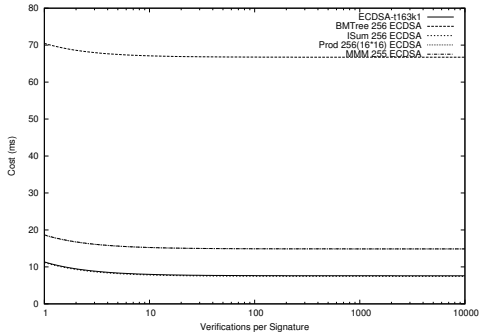


(a) $\mathcal{M}_1$.

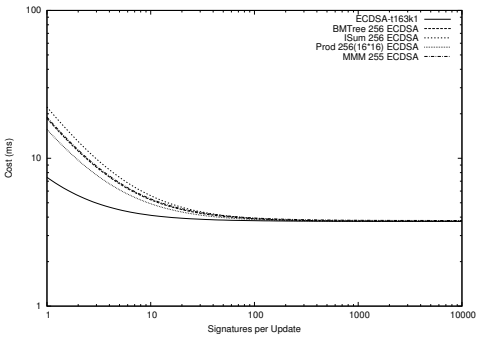


(b) $\mathcal{M}_2^*$.

Figure 3: Performance of short-term forward-secure schemes using RSA, with a maximum update period of 255.



(a) $\mathcal{M}_1$.



(b) $\mathcal{M}_2^*$.

Figure 4: Performance of short-term forward-secure schemes using ECDSA, with a maximum update period of 255.

mum number of periods required from a key is a constant, determined by the usage much like the required key strength. Later in this section we will explore the impact of relaxing this assumption. Table 2 shows the results for one of these combinations, 255 and 256 period FSS keys using short-term security base scheme keys in their construction. Each key is used for its first 255 periods. As before, the other cases with different base key strengths and maximum periods have similar results to those shown, but are omitted due to space constraints.

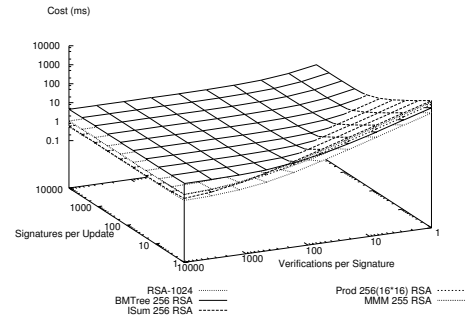
$\mathcal{M}_1$ *and* $\mathcal{M}_2^*$

We first graph $\mathcal{M}_1$ and $\mathcal{M}_2^*$ with the FSS scheme used being the only variable, fixing the base scheme to be either RSA (Fig. 3) or ECDSA (Fig. 4); as we saw in the previous subsection, DSA is always at least an order of magnitude slower than the most efficient scheme, and therefore we do not consider DSA in the remainder of our study. This separation allows the impact on performance due to FSS scheme to be isolated from the impact due to base scheme. Also included in the graphs is the non-forward-secure base scheme used for reference when determining the overhead of FSS.

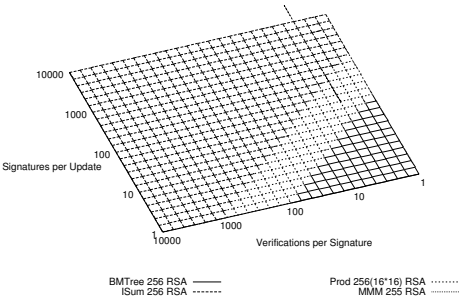
In Fig. 1, we saw that for $\mathcal{M}_1$, RSA was the most efficient signature scheme and for $\mathcal{M}_2$, ECDSA was the most efficient. We therefore begin our analysis of the forward-secure schemes by looking at $\mathcal{M}_1$ for RSA based schemes and $\mathcal{M}_2^*$ for ECDSA based schemes.

As expected, in Fig. 3(a) the schemes separate into three clusters, where within each cluster there is essentially no difference in performance among schemes for $\mathcal{M}_1$. These clusters correspond directly to the number of base signature verifications required by each FSS scheme as described in Section 2. The fastest are RSA itself and Iterated Sum, both of which perform a single verification. MMM and Product both consist of two Iterated Sum keys bound together, so logically they display a cost that is twice that of Iterated Sum. Bellare-Miner Tree, which has to perform verifications proportional to the depth of the tree (but only a single base signature on FSS sign), has the worst performance by a significant amount. The signing costs for all four FSS schemes are nearly identical to that of the base scheme, and have no impact in differentiating schemes. The rate at which the three groups stabilize to a nearly constant cost closely resembles that of RSA, indicating that the FSS scheme itself has little impact on this behavior.

Turning to $\mathcal{M}_2^*$ for ECDSA-based schemes in Fig. 4(b), we see that initially all four FSS schemes perform measurably worse than ECDSA, ranging from twice as expensive for Product to three times as expensive for Iterated Sum. Product, with the fastest update and signing costs when using ECDSA, is the least expensive forward-secure scheme for all values of $\mathcal{R}_2^*$; meanwhile, Iterated Sum, whose weakness is very expensive key generation and updating, is the slowest. Unlike the results for $\mathcal{M}_1$, no clustering of schemes occurs for the second metric. Instead, all four FSS schemes converge to the cost of non-forward-secure ECDSA by 100 signatures per update, after which point they are nearly indistinguishable. This behavior can be attributed to the cost of signing being identical for



(a) $\mathcal{M}_3^*$.

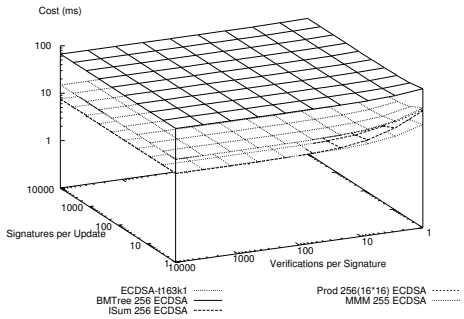


(b) Lowest-cost FSS scheme for $\mathcal{M}_3^*$.

Figure 5: Performance of short-term FSS schemes with 255 periods using RSA.

all four FSS schemes as well as the base scheme, and the update cost being sufficiently amortized.

Because it is impossible to pick one base scheme to use when generating and updating a FSS key and a different one to use for verification, it is useful to look at the other half of the picture for each base scheme to see how much worse than the optimal base scheme it performs. In Fig. 4(a), the same characteristics seen is Fig. 3(a) are still found. The only differences are the overall cost, and the speed at which the clusters stabilize to a near constant cost, both of which are exactly the same as the differences between ECDSA and RSA themselves. As with $\mathcal{M}_1$, some of the differences in Fig. 3(b) for $\mathcal{M}_2^*$ also directly reflect those seen in Fig. 1(b) for RSA and ECDSA. Because of the higher cost of key generation (and hence update) for RSA, the time required for the different FSS schemes to converge to the same cost as RSA is significantly larger than for ECDSA. Unlike the first metric, however, the ordering of FSS schemes changes between Figs. 4(b) and 3(b), with only Iterated Sum as the slowest remaining the same. The reason for this is that while ECDSA has almost the same cost for key generation and signing, for RSA the first of these operations is much more expensive. This switches the advantage from Product, which has a number of key generations but only one signature, to Bellare-Miner Tree, which has more signatures but fewer key generations than any of the other schemes.



(a) $\mathcal{M}_3^*$.

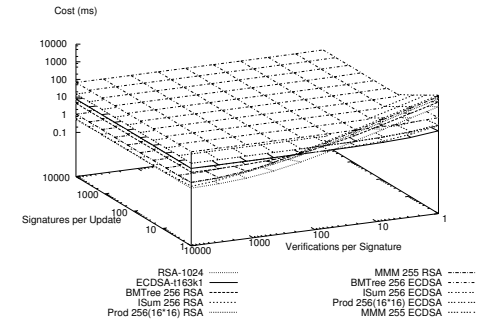
Figure 6: Performance of short-term FSS schemes with 255 periods using ECDSA.

$\mathcal{M}_3^*$

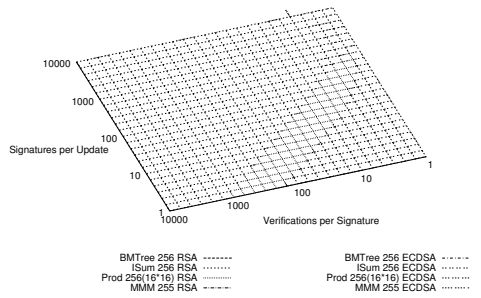
We now arrive at the heart of our analysis. Combining the results for $\mathcal{M}_1$ and $\mathcal{M}_2^*$, we see $\mathcal{M}_3^*$ for the forward-secure schemes plotted in Figs. 5, 6, and 7. We begin by continuing to examine the results with the FSS scheme used being the only variable, and later, we look at all eight configurations together in Fig. 7.

In Fig. 5, we see that at different times, Bellare-Miner Tree, Iterated Sum, and Product each are the most efficient when using RSA as the base scheme. Bellare-Miner Tree starts out as the most efficient for low verification and signature frequencies due to the increased impact C_u has on overall performance in these situations. As either $\mathcal{R}_1$ or $\mathcal{R}_2^*$ grows, Bellare-Miner Tree's performance stays nearly constant while the remaining algorithms quickly improve. Product, which had good performance for both $\mathcal{M}_1$ and $\mathcal{M}_2^*$, briefly takes over as the lowest cost, with MMM and Iterated Sum approximately one and a half times more expensive at the point where Product and Bellare-Miner Tree are equal. Finally, as was the case with traditional schemes, as the parameters grow larger, the results of $\mathcal{M}_1$ dominate. Iterated Sum, with its optimal C_v, becomes the scheme to use. At the crossover point, MMM, Iterated Sum, and Product are all nearly identical in cost but Bellare-Miner Tree is already an order of magnitude slower. At the transition between Bellare-Miner Tree and Product, the difference between FSS and RSA is the greatest. The maximum occurs at one verification per signature and 100 signatures per update, where FSS is four times as expensive as RSA. As the parameters continue to increase, the costs decrease and stabilize with Iterated Sum and RSA having nearly identical performance, MMM and Product half as fast, and Bellare-Miner Tree significantly slower.

For Elliptic Curve DSA, the results are less interesting. Unlike RSA, where C_u is significantly greater than C_s or C_v, all three costs are much closer to one another for ECDSA based configurations. Therefore, even when $\mathcal{R}_1$ and $\mathcal{R}_2^*$ are small, the share of update and signing for each verification never grows large enough outweigh the savings in Iterated Sum from having to perform fewer base verifications. In Fig. 6, Iterated Sum is always the least expensive scheme, and quickly converges to the same cost as ECDSA. MMM and Product are almost indistinguishable from one another, and both converge to twice the cost of Iterated Sum or ECDSA. Bellare-Miner Tree is nearly constant in its cost for all values of $\mathcal{R}_1$ and $\mathcal{R}_2^*$, but that constant is orders of magnitude higher than



(a) $\mathcal{M}_3^*$.



(b) Lowest-cost FSS scheme for $\mathcal{M}_3^*$.

Figure 7: Performance of short-term FSS schemes with 255 periods.

the cost of the rest of the schemes. The worst performance relative to ECDSA occurs initially, where the expensive key generation leads to Iterated Sum costing twice as much as ECDSA.

Finally, we look at the complete picture with both ECDSA and RSA as potential base schemes, seen in Fig. 7. As might be predicted from Fig. 2, when $\mathcal{R}_1$ and $\mathcal{R}_2^*$ are small, the ECDSA based configurations are dominant in Fig. 7, while as either ratio grows larger the RSA based configurations take over. The area over which ECDSA based Iterated Sum is most efficient covers all of the RSA Bellare-Miner Tree region of Fig. 5 and portions of the RSA Product and RSA Iterated Sum regions. When $\mathcal{R}_1$ is less than two, ECDSA Iterated Sum is always the most efficient, which can be seen on the right in Fig. 7(b). As was also the case when examining the base schemes alone, the intersection between RSA based schemes and ECDSA based schemes occurs at a much greater slope than the intersections among different RSA based schemes. It is at this point where performance is worst, with RSA Product 2.6 times as expensive as RSA.

Looking at the applications of FSS described in Section 1, we are now able to determine the optimal FSS configurations to use. For the electronic checkbook, each check will only be verified a limited number of times, possibly as few as once by the user's bank. In order to limit her exposure to forged checks, it is also in the user's best interest to update her private key after writing a check. This describes a scenario where both $\mathcal{R}_1$ and $\mathcal{R}_2^*$ are small. Iterated Sum using ECDSA would be the optimal choice in this environment.

Max Periods	RSA 1024	RSA 1536	ECDSA t163k1	ECDSA t233k1
256	4.47ms	8.60ms	66.7ms	123.4ms
512	5.00ms	9.53ms	74.0ms	137.1ms
4096	6.35ms	12.4ms	94.7ms	176.1ms

Table 3: Average C_v for Bellare-Miner Tree FSS keys of various maximum periods.

For the case of a Certificate Authority, as with traditional signatures $\mathcal{R}_1$ will be very large. $\mathcal{R}_2^*$, because it is per update and not per key, may no longer be as large as in the traditional case. The optimal FSS configuration in this situation is most likely Iterated Sum with RSA keys, which has almost identical verification time to RSA. It is possible that, if both $\mathcal{R}_1$ and $\mathcal{R}_2^*$ are low (e.g. 100 and 5 respectively), the optimal scheme will be Product using RSA instead. However, as we noted above in our analysis of RSA based schemes alone, in this region all three of MMM, Product, and Iterated Sum have nearly identical performance.

The final example use of FSS presented is the signing of receipts by an on-line merchant, with updates performed at the close of business. Here, the number of verifications is very low, as in most cases when there is nothing wrong with the order the receipt will never need verifying. $\mathcal{R}_1$ is very low because of this. The number of signatures, however, may be quite large if the store is busy. As with the electronic checkbook, Iterated Sum using ECDSA is the best choice for this situation.

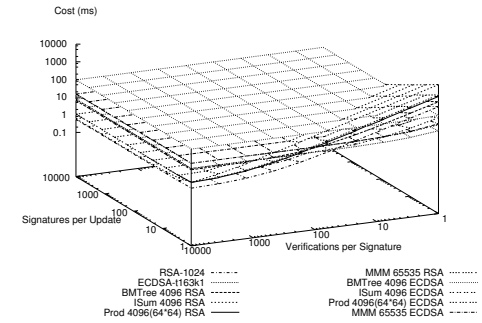
Unknown Maximum Period

Up until this point, we have assumed that the maximum number of periods needed for a forward-secure key is known exactly ahead of time, and examined the performance of different schemes based on this assumption. In many situations, however, the exact upper limit on the number of periods required may not be known. If more periods are required than provided by the private key, a new key must be generated and certified, reducing some of the benefits of FSS schemes over traditional signature schemes.

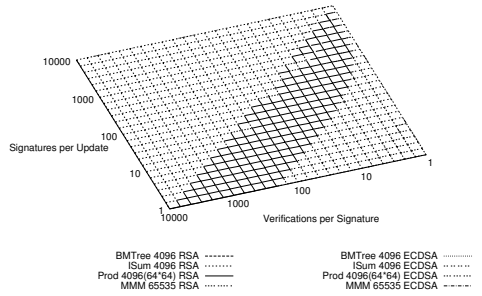
The most straightforward solution to this problem is to be conservative when generating forward-secure keys, and specify a maximum period much larger than what is actually expected to be needed. Depending on the forward-secure scheme used, however, this may impact the performance for the expected range of periods significantly, and may even impact the decision of what FSS configuration to ues. In order to evaluate this penalty we look at how the average values of C_u, C_s, and C_v change as the number of periods used increases for keys of various maximum periods, and re-examine $\mathcal{M}_3^*$ with these new costs.

The results for C_s are not very interesting, as the cost of signing is completely dominated in all four FSS schemes by the single base signature performed. Regardless of how many periods used or what the maximum period of the key is, the cost of signing is the same as the cost for the base algorithm signature.

For Iterated Sum, Product and MMM, the same holds true for the cost of verification. Similar to signing, all three of these schemes require a fixed number of verifications regardless of the current period or the maximum period. Bellare-Miner Tree, however, is dependent on the maximum period (but not the current period) for the number of base verifications it must perform, and therefore C_v is dependent as well. Table 3 shows the average verification cost for the ten Bellare-Miner Tree configurations tested. As expected, the



(a) $\mathcal{M}_3^*$.



(b) Lowest-cost FSS scheme for $\mathcal{M}_3^*$.

Figure 8: Performance of short-term FSS schemes with 4096 periods after 255 periods.

difference in cost from doubling the maximum period while keeping the base scheme fixed is almost exactly the cost of a single base verification.

The impact of maximum period on C_u also depends on the scheme. For MMM beyond the first thirty periods and for Bellare-Miner Tree for all periods there is no real difference in performance between keys of different maximum periods. Iterated Sum and, to a lesser degree, Product (because it contains Iterated Sum keys) are logarithmically related to the period for the performance of C_u. For a situation requiring only 255 periods of a key to be used, the cost of using a 4096 period Iterated Sum key is 4 times greater than using a 256 period key. Product in comparison experiences only a 30-35% slowdown in this situation.

Fig. 8 shows the results of computing $\mathcal{M}_3^*$ assuming 255 periods are used, but with keys having a maximum period of 4096 (or 65535 in the case of MMM). In the figure, the higher C_u for Iterated Sum can be seen by the much greater upward deflection towards the origin for the two Iterated Sum surfaces compared to Fig. 7. This increase in initial cost causes MMM and Product (which have nearly identical initial costs) to become more efficient than ECDSA Iterated Sum when $\mathcal{R}_1$ and $\mathcal{R}_2^*$ are less than 10, and the point where RSA Iterated Sum becomes most efficient to move out significantly compared to Fig. 7. The point where RSA Product intersects ECDSA Iterated Sum also moves further away from the origin due to the greater increase in C_u for RSA based schemes than ECDSA based schemes. As before, over much of the area where Product is least expensive the remaining RSA schemes (except Bellare-Miner Tree) are also nearly equal. This is the cause of the noisy region in Fig. 8(b) where the most efficient scheme changes between several different schemes in a small area.

The impact of these changes in the behavior of FSS keys on their use is minimal. For a CA, the position again lies either in the RSA Iterated Sum region, or in the transitional region where all of the RSA schemes except Bellare-Miner Tree are nearly equivalent. If $\mathcal{R}_2^*$ is less than 5, MMM or Product will likely be slightly faster than Iterated Sum. For the merchant signing receipts, the number of signatures per day (update) remains large enough that ECDSA Iterated Sum is the best choice. The one example situation looked at where the decision on which algorithm to use does change is in the electronic checkbook application, assuming that the maximum number of checks (periods) is not set in advance. In this situation, ECDSA MMM or Product are both able to produce keys with very large maximum periods but with little initial cost or increase to update cost compared to ECDSA Iterated Sum.

In all, even without knowledge of the exact number of periods required the performance of the optimal FSS schemes remains good when compared to the optimal traditional signature schemes. The difference is greatest at the transition between ECDSA-based schemes and RSA-based schemes once again, where FSS is 3.3 times more expensive than RSA alone. As the signatures per update or verifications per signature grow, the FSS cost converges on the traditional scheme's cost as before and there is negligible overhead to FSS.

4. FSS REFERENCE IMPLEMENTATION

While implementing the forward-secure signature library benchmarked in the previous section, several interesting issues were encountered which previous works on forward-secure signatures have not explored due to their theoretical nature. These issues have also not been looked at fully in works on non-forward-secure cryptographic library implementations as they are unique to, or uniquely influenced by, forward-secure signatures. In this section we begin with a brief overview of the `libfss` library's design and architecture, and then look at the issues encountered implementing this design, and the steps taken to overcome them. A more detailed technical description of the library and its implementation can be found in the Tech Report [3].

4.1 Design and Architecture

The `libfss` library is a C library providing a generic interface to forward-secure signature operations as well as implementations of a number of FSS signature schemes. The library uses the OpenSSL [38] cryptographic library to provide RSA, DSA, and ECDSA implementations as well as other cryptographic support functions such as hashing and random number generation. OpenSSL is one of the most widely used open-source cryptographic implementations; it runs on almost all varieties of Unix-like operating systems as well as Windows, MacOS, and many embedded devices. By using OpenSSL for the core cryptographic operations, `libfss` is able to take advantage of over a decade of development and tuning for these elements (as Section 3 showed, performance is largely determined by the performance of the base algorithm). For many platforms, OpenSSL provides highly optimized assembly code implementations of these critical functions. Another advantage to OpenSSL is that it has recently added the ability to transparently use cryptographic acceleration hardware when available, a feature which `libfss` will be able to leverage when possible.

The API of `libfss` is modeled after EVP API used in OpenSSL

for encryption and signature operations. It is a generic API where "key" and "signature" objects contain type information which is used to select the correct implementation to use for operations such as sign, verify, and update. This allows the application using of the library to place all code dealing with different signature algorithms in a single place, when a key is initially generated, and the remaining calls to functions are scheme-independent. By using an API similar to OpenSSL's, `libfss` can be used in existing programs with little modification to the existing code.

One significant difference between the `libfss` design and OpenSSL's EVP design is that the generic interface is used internally as well for `libfss`. While OpenSSL uses a hard-coded list of known signature types, our library uses a callback structure similar to OpenSSL's Engine API (for supporting hardware accelerators) or the BSD Virtual Filesystem Switch (VFS) [29]. Although this design adds an extra layer of indirection in certain situations, it also allows additional FSS schemes to be used without requiring a recompilation of the library. Not only can these new schemes be used with the generic FSS API, but they can be used in the built-in constructions such as Iterated Sum and MMM.

The library currently contains implementations of five forward-secure signature schemes as well as wrappers for the three built in signature schemes that treat them as one-period forward-secure schemes. The schemes implemented are the Tree scheme from [5] and the Sum, Iterated Sum, Product, and MMM schemes from [28], as well as RSA, DSA, and ECDSA.

4.2 Implementation Challenges

4.2.1 Deterministic Key and Signature Generation

For traditional signature schemes, being able to generate the same key twice in practice is seen as a major security flaw. OpenSSL, as well as many other cryptographic libraries, take steps to prevent a user from accidentally doing this through misconfiguration on the the pseudo-random number generator (PRNG) or similar mistakes. For generic forward-secure constructions, those based off of the Sum construction in particular, the ability to deterministicly generate a key multiple times given the same random seed is, however, a necessity. This requirement for reproducibility also extends to signature generation for schemes like DSA, where random numbers are used in signing as well. If signatures are not deterministic then private keys for the Product and MMM schemes, for example, cannot be made deterministic.

In OpenSSL key generation, the primality tests and other operations requiring random numbers draw directly from OpenSSL's internal global entropy pool, which itself is typically filled from system level sources of randomness [39]. Once initialized, OpenSSL's entropy pool cannot be reset to a known value, only updated with additional entropy. For these reasons, there is no way to deterministicly generate a key in this architecture without completely replacing OpenSSL's random number generator with one that can be reset to a specific state. Because this global PRNG is used not just by `libfss` but also potentially by the application itself through the use of other OpenSSL functions, replacing the PRNG altogether is not an acceptable design.

The solution taken by `libfss` is to incorporate the code from OpenSSL for generating RSA, DSA, and ECDSA keys (as well as DSA and ECDSA signatures) into `libfss`, and replacing any calls to OpenSSL's PRNG with calls to our own PRNG in the copies. This internal PRNG is implemented using AES-128 in Counter mode [32]. A seed provided to the function is used as the key for AES, and random numbers are provided by the encrypted conter output.

Although this deterministic key generation is required within certain FSS schemes for correct operation, it would still be unwise to expose it to users who could accidentally generate keys with insecure random seeds. For this reason the top level API does not provide a way to specify the seed to be used, and instead generates a random seed from OpenSSL's PRNG. Internally, when a key or signature needs to be generated, the callback is used and it is possible to specify a seed.

This need for repeatability also raises potential problems for the use of hardware cryptographic accelerators with forward-secure signature schemes. Accelerators that perform entire large-scale operations (such as an entire key generation or signature) on-chip using an internal random number generator would cause similar problems, and likely be unusable for these FSS constructions. On the other hand, accelerators that only provide hardware assistance for cryptographic building blocks such as modular exponentiation or large number arithmetic, would cause no problems.

4.2.2 Secure Deletion of Sensitive Key Material

As previous works have shown, it is exceptionally difficult to securely remove all traces of sensitive information from the numerous locations it might reside [21, 12, 35]. The compiler can optimize away "useless" instructions intended to clear memory before releasing it. The operating system can page a block of memory out to disk, leaving a copy on physical media until overwritten. Physical devices (RAM, magnetic media) can permanently retrain traces of data written to them, even after it is overwritten.

The same precautions and solutions used to protect traditional private keys apply to forward-secure private keys. Keys should never be written to permanent media without first being encrypted using a symmetric encryption algorithm. Whenever sensitive key material in memory is no longer needed, a special low-level 'cleanse' function which works around compiler optimizations should be used. When available, encrypted swap or pining pages containing private keys in memory should be used to prevent keys inadvertently being written to disk unencrypted. Forward-secure keys are no less secure than traditional keys in these regards.

However, in addition to protecting the current key from accidental exposure, FSS implementations must also worry about how to permanently and securely destroy old keying material during update. For the copy in memory, the procedures in the previous paragraph handle this as well. The complication arises with the other, permanent, copies of the key, even when stored encrypted. Unless the key used to encrypt it is destroyed (implying that each period of a FSS key has a unique encryption key), an earlier private key would be recoverable from disk. Because it is often difficult or impossible to completely erase data from disks [21], this can pose a significant risk. Even worse, periodic backups, logging/journaling filesystems, and other common features can all lead to multiple copies of the encrypted key existing. For this reason, containing even the encrypted key is important for ensuring the security of a forward-secure scheme.

It is important to note, however, that this is no different than when many short-lived traditional private keys are used instead of FSS. By having many private keys to protect against exposure, both solutions introduce the new problem of having many private keys to securely manage.

4.2.3 Protecting Against Timing Attacks

A common problem faced when implementing cryptographic systems is that even though the implementation itself may be correct and secure against the primary threat, it may inadvertently be vulnerable to *side-channel* attacks that can expose sensitive data in un-

expected ways. One common type of side-channel attack signature schemes often face is timing attacks [25, 24]. Timing attacks are possible whenever an operation is performed in an automated and interactive fashion, such as protocol negotiation or operations performed by a smart card. By choosing specific inputs and measuring the time between request and reply, it is possible for an attacker to infer information about the private key that compromises security.

There are two main varieties of timing attacks that have been discovered in the past: those inherent to the algorithm itself, and those due to a particular implementation of the algorithm. We examine the impact of each of these on `libfss` in turn.

The first class of timing attacks arise due to the fact that with some algorithms, the amount of computation required for an operation varies significantly based on the input and private key together. By carefully selecting inputs, an attacker can use this to determine the private key a bit at a time [25]. The FSS schemes in `libfss` are all of the generic type and do not perform any key dependent operations themselves, so they are only vulnerable to these types of attacks if the base scheme used is. For other FSS schemes described in Section 2 (but not implemented in `libfss` currently) that are complete cryptographic algorithms and not generic constructions, this type of attack may apply directly.

For functions that are dependent on the private key in their performance, there are sometimes alternate ways of performing the necessary computation that do not depend on the private key. RSA "blinding" is an example of this, and `libfss` supports this technique to protect private keys using RSA as the base scheme. RSA blinding uses a random number and splits the computation into two parts each using that number. By doing so, any correlation between private key and execution time is removed.

When it is not possible to redesign the algorithm to remove the correlation between private key and execution time, another technique that increases the amount of work required by the attacker is quantization [7]. Quantization works by padding the computation time until it is a multiple of a fixed quantum before returning the result to the user. Unless the quantum is larger than the operation could ever take, this countermeasure only adds noise to the attacker's measurements and does not provably remove the attack [25]. Nonetheless, in practical terms it does significantly increase the number of queries required.

To protect against these attacks, the FSS wrapper for RSA provides an option to enable a feature known as "blinding" that adds a random element to the computation removing any correlation between the private key and execution time. For other base algorithms where timing attacks exist, the library does not contain support for quantizing internally, but software such as Matt Blaze's [7] library can be used with `libfss`.

The second type of timing attack is caused by the implementation, typically with the way errors are handled. If a complex operation aborts as soon as an error is detected, it provides an indication as to which part of the input was invalid through the elapsed time. For forward-secure verification for example, this could be used to infer which signatures in a chain are valid and invalid by introducing intentional corruptions and timing the verification. This could then be used to isolate a single base signature and key to attack with the first type of timing attack.

The solution to this type of attack is to postpone returning an error as late as possible. `libfss` accomplishes this by continuing after all non-fatal errors, and returning an error if any stage of the operation fails. In addition, all operations for which failure prevents continuing, such as memory allocation and conversion of keys from encoded binary forms to internal structures, are moved as far forward in the operations as possible so as to detect these errors before any cryptographic operations have been performed on the input. With these two steps, `libfss` does not leak any information about which portion of a signature is invalid on error.

5. CONCLUSION

In this paper we have explored the practical performance characteristics of forward-secure signature schemes. In the process, we define a new framework for comparing signature schemes which takes into account the application environment in computing an amortized cost for basic operations. We use this tool to compare several different FSS schemes built using generic constructions, as well as several non-forward-secure signature schemes used as bases for these constructions. We use our performance metrics to examine a number of example uses for forward-secure signatures, and provide recommendations as to the optimal FSS scheme and configuration to use for each of these applications.

Our empirical study of FSS performance shows that, despite key generation and update operations which are significantly more expensive than non-forward-secure equivalents, the performance of FSS is actually quite competitive if correctly used. In environments such as a Certificate Authority, the overhead of FSS is almost nonexistent when costs are amortized. The greatest difference in performance between FSS and traditional signature schemes occurs when there are few signatures and verification made by each key; even in these cases, FSS performs only two to four times slower, not hundreds or thousands times slower as the raw operation costs might indicate.

These results show concretely that forward-secure signatures are very practical. Many applications which currently use traditional signatures could be switched to using forward-secure signatures with little impact on performance, but an enormous impact on the amount of inconvenience faced on key exposure. To further help the adoption of forward-secure signatures, the reference implementation used in this study will be released under an open-source license and available, along with developer documentation, from the project web site at `http://anonymized/`.

There are several directions for future work based on this study. We have only looked at the performance of these generic FSS constructions using software implementations of the base signature schemes. Many applications are now relying on hardware-based cryptographic co-processors when making traditional signatures, and the role these devices play in FSS needs to be explored. Our comparison also focused only on generic constructions due to their desirable property of being built upon well known and well trusted traditional signatures. Nonetheless, the performance of the other schemes described in Section 2 such as Bellare-Miner and Itkis-Reyzin are also of interest, and in the future we hope to expand our FSS reference implementation to include these schemes.

6. REFERENCES

[1] M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. *Advances in Cryptology – ASIACRYPT 2000, Lecture Notes in Computer Science*, 1976:116–129, Dec. 2000.

[2] R. Anderson. Two remarks on public-key cryptology *From Invited Lecture, Fourth ACM Conference on Computer and Communications Security (April, 1997)*. `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-549.pdf`.

[3] Anonymized. Design and implementation of a forward-secure signature reference library. Technical report, Anonymized, 2003.

[4] ANSI X9.62-1998. Public key cryptography for the financial services industry: Rhe elliptic curve digital signature algorithm (ECDSA), 1998.

[5] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. *Advances in Cryptology – CRYPTO '99, Lecture Notes in Computer Science*, 1666:431–448, Aug. 1999.

[6] M. Bellare and B. S. Yee. Forward-security in private-key cryptography. In *Topics in Cryptology - CT-RSA '03, The Cryptographers' Track at the RSA Conference 2003*, 2003.

[7] M. Blaze and J. Lacy. Simple Unix time quantization package, 1995. `http://islab.oregonstate.edu/documents/People/blaze/quantize.shar`.

[8] J. N. Bos and D. Chaum. Provably unforgeable signatures. *Advances in Cryptology – CRYPTO '92, Lecture Notes in Computer Science*, 740:1–14, 1993.

[9] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. In *Proc. of the thirtieth annual ACM symposium on Theory of computing (STOC '98)*.

[10] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *Proc. of the 21st Annual IACR Eurocrypt conference (EUROCRYPT '03)*, 2003.

[11] Certicom Research. SEC 2: Recommended elliptic curve domain parameters, Sep. 2000. `http://www.secg.org/secg_docs.htm`.

[12] G. D. Crescenzo, N. Ferguson, R. Impagliazzo, , and M. Jakobsson. How to forget a secret. *STACS '99, Lecture Notes in Computer Science*, 1563:500–509, 1999.

[13] T. Dierks and C. Allen. The TLS protocol. RFC 2246, IETF, January 1999.

[14] W. Diffie and M. E. Hellman. Multiuser cryptographic techniques. In *AFIPS Conference Proceedings*, volume 45, pages 109–112, 1976.

[15] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes, and Cryptography*, 2(2), 1992.

[16] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *Proc. of the 20th Annual IACR Eurocrypt conference (EUROCRYPT '02)*, 2002.

[17] Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. In *Proc. of the 6th Annual International Workshop on Practice and Theory in Public Key Cryptography (PKC '03)*, 2003.

[18] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. *Advances in Cryptology - CRYPTO '86, Lecture Notes in Computer Science*, 263:181–187, 1986.

[19] L. C. Guillou and J.-J. Quisquater. A "paradoxical" identity-based signature scheme resulting from zero-knowledge. *Advances in Cryptology – CRYPTO '88, Lecture Notes in Computer Science*, 403:216–231, Aug. 1988.

[20] C. Gunther. An identity-based key-exchange protocol. In *Proc. of the 7th Annual IACR Eurocrypt conference (EUROCRYPT '89)*, 1989.

[21] P. Gutmann. Secure deletion of data from magnetic and solidstate memory. In *Proceedings of 6th USENIX UNIX Security Symposium*. USENIX Association, July 1996. San Jose, CA.

[22] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. *Advances in Cryptology – CRYPTO '01, Lecture Notes in Computer Science*, 2139:332–354, Aug. 2001.

[23] G. Itkis and L. Reyzin. SiBIR: Signer-base intrusion-resilient signatures. *Advances in Cryptology – CRYPTO '02, Lecture Notes in Computer Science*, 2442, Aug. 2002.

[24] B. Kaliski. Timing attacks on cryptosystems. *RSA Bulletin*, (2), January 1996.

[25] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Advances in Cryptology – CRYPTO '96, Lecture Notes in Computer Science*, 1109:104–113, 1996.

[26] A. Kozlov and L. Reyzin. Forward-secure signatures with fast key update. In *Proc. of the 3rd International Conference on Security in Communication Networks (SCN '02)*, 2002.

[27] H. Krawczyk. Simple forward-secure signatures from any signature scheme. In *Proc. of Seventh ACM Conference on Computer and Communications Security*, pages 108–115, Nov. 2000.

[28] T. Malkin, D. Micciancio, and S. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Proc. of the 20th Annual IACR Eurocrypt conference (EUROCRYPT '02)*, 2002.

[29] K. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Longman Inc., second edition, 1996.

[30] R. C. Merkle. A digital signature based on a conventional encryption function. *Advances in Cryptology – CRYPTO '89, Lecture Notes in Computer Science*, pages 428–446, 1989.

[31] National Institute of Standards and Technology. Digital signature standard, FIPS 186-2, 2000.

[32] National Institute of Standards and Technology. Advanced encryption standard, FIPS 197, 2001.

[33] NESSIE consortium. Portfolio of recommended cryptographic primitives, February 2003. `http://www.cryptonessie.org`.

[34] H. Ong and C. P. Schnorr. Fast signature generation with a fiat-shamir-like scheme. In *Proc. of the 8th Annual IACR Eurocrypt conference (EUROCRYPT '90)*, 1990.

[35] N. Provos. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium*, pages 35–44. USENIX Association, Aug. 2000. Denver, CO.

[36] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

[37] D. X. Song. Practical forward secure group signature schemes. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS '01)*, 2001.

[38] The OpenSSL Group. OpenSLL, May 2003. `http://http://www.openssl.org/`.

[39] J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL*. O'Reilly & Associates, Inc., 2002.

[40] M. J. Wiener. Performance comparison of public-key cryptosystems. *CryptoBytes*, 4(1), Summer 1998.