Practical Strong Security for Mobile Handheld Devices

Matt England, Axel Garcia-Peña, Sugih Jamin
moengland@umich.edu, axel.garcia_pena@supaero.fr, jamin@eecs.umich.edu

The University of Michigan

Abstract. We present a method for constructing signature schemes for use with mobile handheld devices that mitigates the risk of an attacker forging signatures using key material garnered from a lost handheld. This scheme is forward-secure, meaning that signatures created before a breach are still valid, and server-assisted, meaning that a separate untrusted server must assist the device in signing, thereby protecting against offline dictionary attacks. We also present our experience with implementing these types of signature schemes for Java-enabled mobile devices, and the results of our experiments with this implementation.

1 Introduction

Mobile networked devices such as smart phones and personal digital assistants (PDAs) present some unique challenges for creating secure authentication schemes. These devices are generally much less powerful than a typical desktop computer, with relatively limited memory and CPU resources. Depending on the contract a device's owner has with their service provider, they also may pay a per-kilobyte price for network transmission. Importantly, any key material used for creating digital signatures or encrypting data stored on a mobile device is necessarily at greater risk for exposure: It is much easier to lose one's cell phone than one's workstation.

These characteristics create a hazardous environment for the designer of a security mechanism for use on mobile devices. Consider the scenario in which one must generate digital signatures on a smart phone, using key material stored on the phone's internal memory. This sensitive key material would be stored in an encrypted form and "unlocked" by the user when a signature must be generated - typically by entering a personal identification number (PIN) or password. Now, suppose we implement the above scheme using RSA [14] signatures where the on-device key material is secured using a symmetric key scheme such as AES [1]. If an attacker captures the device, and can read the encrypted key material directly from stable storage, he can mount a dictionary attack on the user's PIN. If he is successful, he can forge the user's digital signature. Moreover, given the difficulty of keying in a long passphrase on a typical phone keypad, the user passphrase is likely to be short and therefore susceptible to attack. Using a typical public key revocation scheme, the revocation of the user's key will result in *all* of his signatures becoming invalid - even those created before the device was compromised.

In this paper, we present a new approach to enabling secure signatures on mobile devices that hardens these devices against the type of attack illustrated above. Our approach combines two techniques: *server-assisted* signature schemes and *forward-secure* signature constructs. We have implemented our schemes as a Java security library for use on J2ME capable devices, and deployed it to several such devices. We discuss our experiences with this software, and the results of our experiments.

Server-assisted schemes, as the name suggests, utilize a separate server in order to create valid signatures. They work by splitting key material into two pieces (one for the device, one for the server) and requiring cooperative computation to create signatures. The signature server is untrusted, in the sense that even if it is compromised, it can learn no information that would help it forge user signatures.

Forward-secure signature schemes apply a special *evolution* operation to signing keys in order to provide the property of *forward security*, namely: Once a key used to sign a message during time period i has been evolved, signatures for time period i may no longer be created, either by the "rightful" user or an attacker who compromises sensitive key material. Thus, once an attacker compromises a forward-secure scheme's

key material, only *future* signatures become suspect. Typical forward-secure schemes have a limit on the number of time periods supported by a given key; traditional schemes can therefore be viewed as degenerate forward-secure schemes with a lifetime of a single time period.[1]

Our approach utilizes methods for composing forward-secure signature schemes to create new schemes with longer lifespans. With composition operations and standard signature schemes as our building blocks, we can recursively apply operators to generate schemes of arbitrary length. Thus we arrive at our goal of forward-secure, server-assisted signature schemes.

Forward-secure, server-assisted signature schemes provide protections for a variety of different types of applications. Consider a program where users report gas prices to a central database via their mobile phones when they are at the pump. That is, if Alice stops to get gas on Monday morning and pays $3.19, she can notify a reporting service of the price she paid. If Bob later checks the service looking for the cheapest gas within a certain area, he might find Alice's reported price to be the best. Now, suppose that Alice is a busy woman, and does not report the price she paid merely out of the goodness of her heart - suppose the reporting service offers a reward system for people who report prices (e.g., free gas), and penalizes users who upload false updates (as unscrupulous gas station owners might be tempted to do). In that case, Alice's updates must be signed so that she can be credited for her report, and she needs a way to report the loss of her mobile phone so that nobody can cause penalties to be applied to her account. Forward-secure server-assisted schemes are an ideal solution: They make it difficult for an attacker to impersonate Alice, even if Alice loses her phone, and Alice can rest easy knowing that rewards earned before the loss of her device are safe. So long as the reporting service trusts the signature service, and the signature on an update it receives is valid, it can be sure of the authenticity of the message.

Alternately, consider an information-gathering system deployed amongst government agencies both local and federal. Users - that is, police officers, paramedics, Federal Emergency Management Agency agents, etc. - post reports and notes to their individual departments via a mobile device. These data are signed by the end-user, and marked with a classification code. An aggregation service sifts through the data of each department and propagates some of it up the chain of command to decision makers at a higher level. The aggregation service makes choices based on the classification code tagging each piece of data, and checks the validity of all data via signature verification. This scenario calls for server-assisted forward-secure signatures, as all parties - the departments, the aggregator, the higher-level decision makers - must trust a signature scheme without necessarily controlling it (that is, the local police department does not control the signature service, though perhaps the high-level authorities would).

In Section 2, we present an overview of related work. Section 3 outlines our approach to creating forward-secure server-assisted signature schemes, and describes the building blocks of this design. Section 4 describes our implementation of forward-secure server-assisted signature schemes in Java, and some optimizations adopted from our experience with this implementation. Section 5 presents the results of some performance measurements on our implementation in its several incarnations, and Section 6 describes our plans for further work. Section 7 concludes the paper.

2 Related Work

The problem of designing a forward-secure digital signature scheme was introduced by Ross Anderson in a lecture at the ACM CCS conference [2]. Bellare and Miner [3] formally proposed the first forward-secure signature scheme. Subsequently, Malkin, Micciancio, and Miner [12] proposed several techniques for composing digital signature schemes to create forward-secure schemes: The *Sum* and *Product* operators that we employ, and the MMM scheme built from those operators. As their names suggest, the Sum and Product operators allow one to combine two forward-secure signature schemes with lifetimes of A and B time periods into one with $A + B$ and $A \times B$ time periods, respectively. One may therefore create schemes of any length through recursive application of these operators.

MacKenzie and Reiter [11] propose two server-assisted public key cryptography schemes supporting instant key revocation. One scheme, called Shared RSA (SRSA), uses secret sharing to divide an RSA key's

[1] Speaking of "the lifetime of a scheme" is really a shorthand for the lifetime of its keys.

private exponent into two pieces - one for the device and one for the server. The public key is the usual RSA public key, meaning that signature verification works exactly the same as standard RSA. In this work, we adopt a modified version of SRSA as one of our building blocks for achieving a forward-secure server-assisted signature scheme. Due to the general nature of the composition methods used in the construction of the forward-secure signature (that is, Sum and Product), we suspect that other shared-secret schemes could also be adopted for use where we employ SRSA.

The scheme proposed by MacKenzie and Reiter can be viewed as a particular type of threshold cryptosystem [6] [8] [10]. In general, (l, k, n) threshold signature schemes split private key data among n parties, requiring a quorum of k shares to sign messages, and are resistant to the exposure of any l shares of the private key. Most often, it is the case that $l = k..1$ (that is, exposing k shares allows signatures to be forged, but any fewer than k exposures does not). Thus, SRSA is a (1, 2, 2) threshold signature scheme. General threshold signature schemes, with $k < n$ and $n \geq 3$, are not of particular use for the goal of protecting mobile devices from dictionary attacks, as we are interested in signatures *always* requiring the involvement of the mobile device - not just any k participants.

Our scheme bears some similarity to digital financial mechanisms, insofar as one might view the signature server as controlled by a bank, and the messages being signed as debit authorizations. Electronic fund transfer systems can be divided into two general categories - electronic cash schemes (e.g., [5] [7]) and electronic checking schemes (e.g., [13] [15]). The former transfer units of electronic money, called coins, in an untraceable or anonymous way. The latter debit a user's account in a decidedly traceable fashion. Indeed, our signature schemes could be incorporated into digital payment systems in place of standard signature schemes, providing the additional benefits of secret sharing and forward-security.

3 Basic Scheme Design

We begin this section with a description of the building blocks we will use to create forward-secure server-assisted signature schemes: the Sum operator, the Product operator, and SRSA. We will then explain how we use these tools in conjunction with one another. We highlight only the relevant facts about these building blocks; for full details, see the papers by Bellare et al. [3] and MacKenzie and Reiter [11].

3.1 Building Blocks

The Sum Operator The Sum operator, as the name suggests, combines two forward-secure schemes with lifetimes A and B into a new scheme with $A + B$ time periods. It essentially works by using the first scheme until it expires, then switching to the second (and throwing away the private key data of the first, thereby making it impossible to sign messages for previous time periods). The public key of the overall scheme is a cryptographic hash of the public keys of the two subschemes, and the private key is all the key data of the underlying subschemes. In order to save space, the B scheme's key material is not actually stored while A is still in use; instead, the random seed used by the pseudorandom number generator that generated the B scheme's key is stored. When the time comes to switch to B, its key is regenerated from the seed.

Recall that traditional signature schemes are really degenerate forward-secure schemes with lifetimes of a single time period. Thus, we could create a two-period forward-secure signature scheme by applying the sum operator to two instances of, say, RSA. A four-period scheme could be created by creating a tree structure of depth 2 where the internal nodes are Sum operators and the leaves are instances of RSA. One quickly realizes that a scheme of any length can be created given enough instances of RSA and the Sum operator.

The Product Operator In a similar vein, the Product operator takes two schemes with A and B time periods and produces a new scheme with $A \times B$ time periods. In a nutshell, it does this by creating an entire instance of the B scheme for every time period of the A scheme; thus there are A copies of a B-length scheme, producing a total lifetime of $A \times B$. The public key of the new scheme is simply the public key of A, and the public key of each B scheme is signed for the period of the A scheme's lifetime to which it corresponds.

Signatures are basically simply signatures under the B scheme, concatenated with the certificate of the B scheme signed by the A scheme. Like the Sum operator, the Product operator stores random seeds that allow the regeneration of larger constructs, in order to *save* space.

Obviously, the Product operator is only useful if the schemes upon which it operates have lengths exceeding one; this is why we need the Sum operator. Our basic strategy is to start by applying the Sum operator recursively as described above, then to use the Product operator one or more times on sufficiently large Sum trees to generate the final scheme.

SRSA Shared RSA, or SRSA, is a variant of the RSA algorithm wherein a second party is needed to perform signature operations. It works by randomly dividing the user's private exponent d into two pieces and encrypting one under the public key of the signature server, then destroying the plain-text copy (see figure 1). The client, who holds the unencrypted share, cannot compute an RSA signature without the aid of the server (the only entity capable of decrypting the encrypted share). The split is performed in such a way that the mobile's share, d_m, and the server's share, d_s, sum to d mod $\phi(N)$, where N is the modulus:

$$d_s + d_m = d \bmod \phi(N) \ .$$

Then both parties perform a partial signature of the message m by computing m^{d_m} mod N and m^{d_s} mod N respectively; the client combines the two partial signature by multiplying, obtaining:

$$m^{d_m} m^{d_s} = m^{(d_m + d_s)} \equiv m^d \bmod N \ .$$

Of course, m^d mod N is a standard RSA signature, implying that SRSA signatures are no different than standard RSA signatures from the point of view of the verifier - the same procedure is used to check both.

Fig. 1. The server key share and disable token are encrypted under the server's public key to create the client ticket. The plaintext server key share is then erased.

SRSA adds the ability for instantly revoking a key by also encrypting the hash of a randomly-chosen token value with the server's share of the key - the encrypted tuple is called a *ticket*. To disable a key, the user reveals the input to the hash to the server, which then lists the ticket on the blacklist. When the client holding the blacklisted key sends its ticket to the server, the server refuses to aid the client if the ticket is on the blacklist.

Our goal is to create a signature scheme which is both server-assisted (to allow instant revocation) and forward-secure (to protect past signatures). We do this by applying the Sum operator to some combination of RSA and SRSA schemes, then applying the Product operator.

3.2 Combining the Building Blocks

Consider, for example, a sixteen period scheme created by composing two four-period schemes via the Product operation, where each of these four-period schemes is created using the Sum operation. One can view this scheme as a tree such as that depicted in figure 2. At the leaves of the tree are instances of one or more traditional schemes. For each gray leaf, an entire right subtree of the Product node will be created. For example, for periods 1 through 4, the leftmost gray leaf is used to sign the black subtree's public key, and

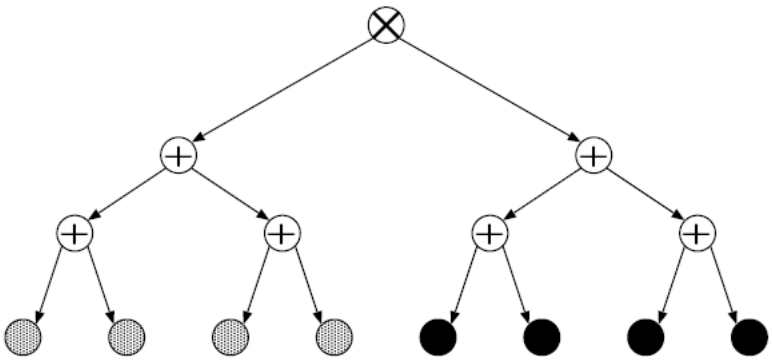

Fig. 2. A 16-period forward-secure signature scheme instance. The gray instances are used to sign the public keys of the right hand (black) tree.

the black subtree is used to sign the message (e.g., period 3 uses the third black key from the left). Upon transitioning to period 5, the gray subtree advances to the second gray leaf, the right subtree is generated anew, and the second leaf is used to sign the new instance of the right subtree. The question we are interested in is this: What traditional schemes should be used for each of the leaves? The best answer to this question is the one that obtains the most efficiency without sacrificing security.

One may simply decide to use SRSA for all of the leaves - this will guarantee that the server is needed for all signature operations. However, since the gray leaves are used only to sign copies of the right-hand (black) subtree when moving from one time period to another (as opposed to when creating signatures for the overall scheme), they need not be instances of SRSA. So long as server assistance is required for every signature operation of the Product scheme, no signature can be created without checking with the signature server. Since the Product operator's signature is really just a signature of its right-hand scheme, the left subtree of the Product operator need not be server-assisted. Thus, the gray instances can be regular RSA, while the black instances must be server-assisted.

Additionally, in the event that the tree's key must be revoked, we would like to revoke *all* of the individual SRSA keys contained within the overall tree key at once. We therefore are careful to use the same disabling token in each SRSA ticket contained in a tree, rather than having to remember a separate disabling token for each instance of SRSA used as a leaf of an operator tree. We also modify the server to blacklist the tokens instead of the tickets, as there is now a one-to-many relationship between a given disabling token and the tickets that contain it, and we want only one blacklist entry for all such tickets.

4 Implementation and Optimization

We have created an implementation of SRSA and the Sum and Product operators in J2ME (Java 2 Micro Edition, the version of Java that is generally supported by smart phones). The implementation utilizes the BouncyCastle [4] cryptography API for its implementation of various well-known primitives such as SHA-1 and RSA.

Our initial implementation ran into several problems. Our assumption is that keys are generated offline (since phones have relatively slow processors), then transferred to the smart phone. The phone would then be responsible for updating keys as time advanced. Since J2ME is a subset of full-fledged Java, it does not support certain objects one might take for granted. For example, it does not contain the objects *java.math.BigInteger* or *java.security.SecureRandom*. The BouncyCastle API provides implementations of both.

J2ME also does not support the object serialization API of standard Java (i.e., *java.io.Serializable* and related interfaces and classes). Therefore, our initial implementation used a homegrown serialization mechanism to convert in-memory key objects to byte streams and back again. This mechanism essentially encoded

the class name of an object in a header section before the serializations of its fields. After some testing, we realized that these class names contributed significantly to the storage overhead, and modified our serialization mechanism to use short codes as stand-ins. The performance section that follows shows the results of this optimization.

Our initial implementation also kept some redundant data which could be recomputed instead of stored (e.g., the hash of two pieces of data as well as the two pieces of data themselves). Since it was not clear what the relative trade-off was between storing redundant data and recomputing it later, we tried it both ways. The performance results in the next section show the extra overhead of storing redundant data.

Finally, our initial implementation suffered from relatively high costs for updating the key for use in the next time period. Recall that the Sum and Product operators reduce storage overhead by storing random seeds used to generate subkeys instead of the subkeys themselves. Consider the simple construct of a Sum operator joining two RSA instances. When we transition to the second time period, we must regenerate the second RSA key. The actual method for doing this is to seed the PRNG *java.security.SecureRandom* with the value originally used during generation of the overall key. The PRNG is then repeatedly queried for a potential prime number of a specific bit-length until a suitable candidate is found (i.e., one that is probably prime with a given certainty) - a replay of the process used during key generation when the second key was generated for the first time. We realized that we could speed up the key update operation at the cost of storage space by memorizing the internal state of the PRNG just before it made its correct guess for each prime, instead of simply storing the seed that led to some n incorrect guesses before the correct one. In other words, instead of storing the seed that would lead the PRNG to generate some number of values, of which most were thrown away, we could instead store the useful values. The performance impact of this optimization is shown in the next section.

5 Experiments and Performance

In this section we present the results of our studies on the performance of our implementations of forward-secure server-assisted signature schemes. Our implementation allows arbitrary configurations of Sum and Product operators; in general, thus far, we have focused on studying configurations that have at most one Product operator. We first show the inherent overhead of using SRSA versus regular RSA to give the reader an idea of the speed with which mobile phones can perform cryptographic operations and access the network. We then go on to discuss the effects of the optimizations outlined in the previous section on the performance and size requirements of a forward-secure server-assisted signature scheme.

5.1 SRSA vs. RSA

The testing device used for comparing SRSA to RSA on a real device was a Nokia 6260 smart phone, which runs a 123-Mhz 32-bit RISC CPU (ARM-9 Series). To give some perspective, the next generation of Nokia phones (the 668x series) use similar processors that run at 220Mhz, thus a significant speedup can be expected on future devices.

Table 1 shows the results of testing SRSA's performance against that of RSA on the mobile device. Not counting the network overhead of SRSA, RSA is roughly five times faster than SRSA. Counting the network overhead, SRSA is two orders of magnitude slower, which is obviously less than desirable. However, in absolute terms, 2.3 seconds per SRSA signature is acceptable if signatures need not be calculated very frequently.

5.2 Optimization Performance

The figures in this section refer to three "versions" of our implementation: Alpha, Tiny, and Fast. Alpha was our original implementation with none of the optimizations from the previous section. Tiny is the implementation that avoids storing redundant data and uses the second, more space-efficient serialization method outlined earlier. Fast is the same as Tiny except that it also memorizes seed values for the PRNG to

make transitioning from one time period to another faster. We show the results of using RSA (not SRSA) as the basic building block for two reasons: (1) the space overhead of an SRSA key is larger than the equivalent RSA key by a *fixed* amount, and (2) due to the network overheads associated with SRSA, iterating a test even a hundred times on a mobile device makes gathering test results in a timely fashion difficult. Given that, aside from the network overhead during signing, SRSA and RSA are nearly identical, we choose to use RSA everywhere.

Figure 3 shows how three variants of our implementation compare to each other and the theoretical minimum in terms of key size. As expected, Tiny is smaller than Alpha, requiring only from 8 to 15 percent more space than the theoretical limit, which is the number of bytes of storage taken by an Alpha key without any metadata (e.g., class serialization codes). The Fast algorithm quickly explodes under the requirement to store so many snapshots of the state of the PRNG before it successfully guesses a prime. This result is not surprising; the question is, in real terms, can we afford to expend the extra storage in exchange for speed? At least for relatively short-lived schemes, the overhead from memorizing the state of the PRNG is still less than the unoptimized Alpha version.

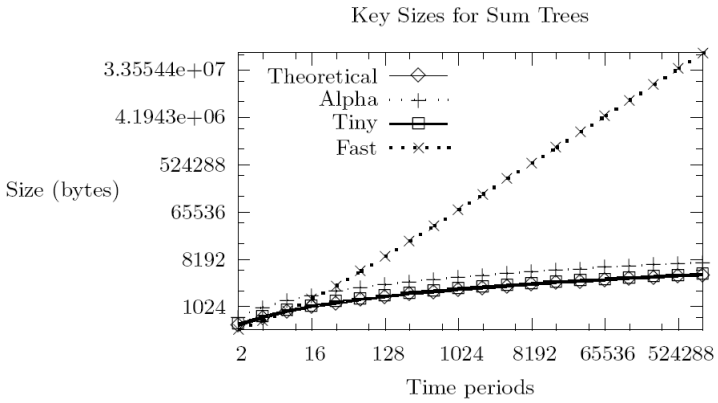

Fig. 3. Key sizes for Sum trees with RSA at the leaf level. Both axes are log-scale.

Figure 4 shows how the signature sizes of our three variants compare. The original implementation, Alpha, grows large as the overhead of maintaining redundant data and long class names grows. The other two implementations have the same signature sizes as one another, with a small gap between them and the theoretical minimum; this is due to the necessity of storing some metadata for serialization purposes.

Figure 5 shows how the introduction of the product operator affects the key size of a scheme. Without space-saving optimizations, the key size of the Product-of-Sums scheme dominates that of an equivalent Sum-only scheme. However, with the optimizations, the Product operator provides significant space reduction over Sum-only based constructs. Again, the Fast storage space grows quickly, suggesting that the speed

Table 1. Timing results comparing SRSA to RSA

	Average	Std. Dev.	Max	Min
RSA KeyGen	18.238s	11.903s	40.781s	6.625s
SRSA KeyGen	17.617s	9.969s	40.140s	7.125s
RSA sign	90ms	7.1ms	94ms	78ms
SRSA sign	2.380s	187ms	2.797s	2.109s
SRSA sign (w/o net)	542ms	128ms	1.125s	468ms

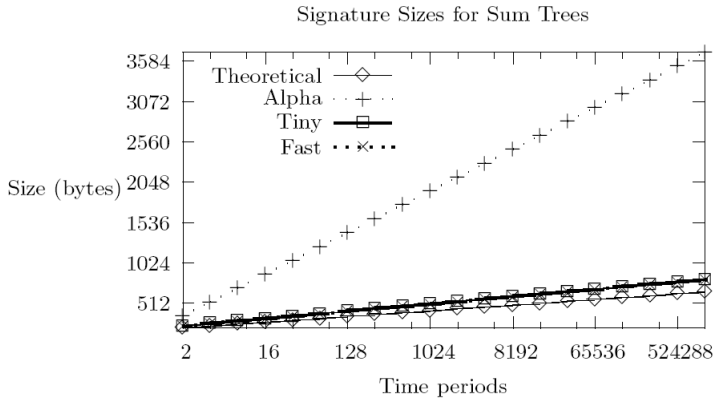

Fig. 4. Signature sizes for Sum trees with RSA at the leaf level. Only the x-axis is log scale.

optimization it provides comes at a significant cost. We now turn to an examination of just how much speedup the Fast optimization provides.

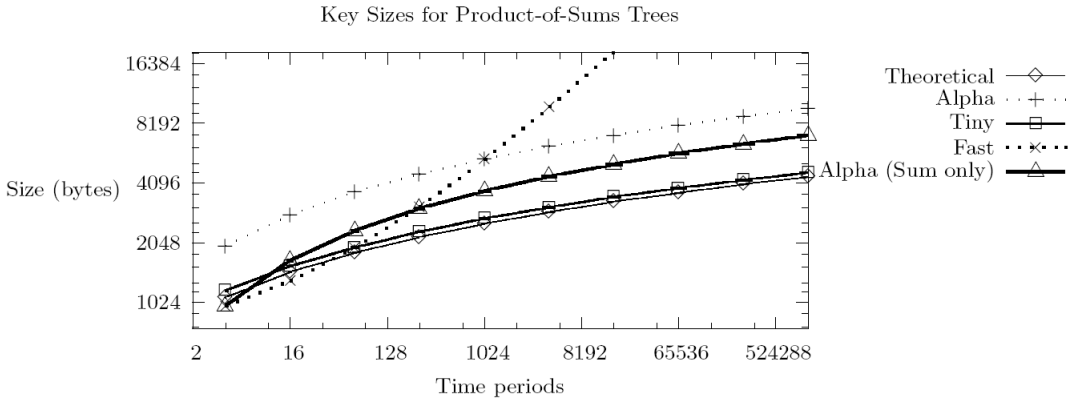

Fig. 5. Key sizes for Product-of-Sum trees with RSA at the leaf level. Both axes are log scale. The key sizes for Sum-only Alpha trees are shown for comparison.

To measure speed, we tested our three variants on a Qtek 8310 smartphone running Windows Mobile 5.0 on a TI IMAP 850 (200MHz) processor. As expected, the validation and signing speeds of all three variants were virtually identical; validation takes 25-28ms, and signing takes 65-75ms. The real difference comes in the update speeds, shown in figure 6. As expected, Fast outperforms both Alpha and Tiny by a significant margin - about 60 percent - cutting update time from the neighborhood of 25 or 30 seconds to approximately 10 seconds. Alpha and Tiny are basically identical; the variations present in the figure are statistically insignificant (i.e., both lines lie within one standard deviation of the other).

The results from this section show that our optimizations for speed, although relatively costly in terms of space, are most likely worth the trade-off for constructs with fewer than one thousand time periods. Though mobile phone memories are limited by the standards of more conventional computing platforms, key sizes of 4096 bytes or so are not a particular strain on modern devices.

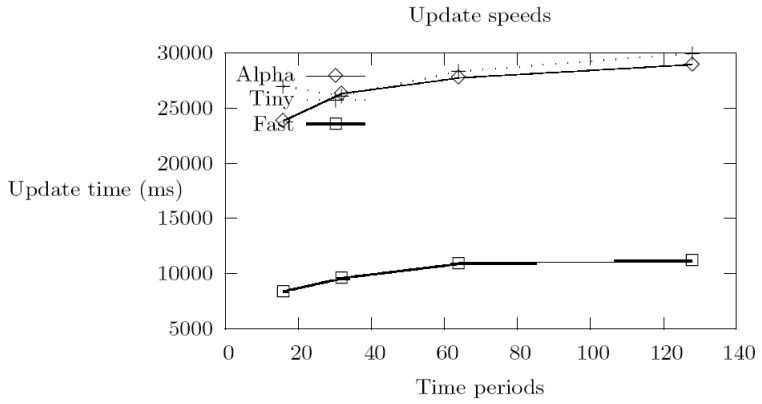

Fig. 6. Update speeds for Product-of-Sum trees with RSA at the leaf level.

6 Future Work

There are several directions in which we would like to take our work in the area of forward-secure server-assisted signature schemes.

The first direction would be to experiment with other base signature schemes to try to effect an increase in performance. Cronin et al. [9] show that ECDSA-based schemes outperform RSA-based schemes when used as the basis for Product and Sum constructions. Though we are not sure if any server-assisted elliptic curve cryptosystems exist, or can be constructed, it seems likely that improved performance can be gained by substituting ECDSA for RSA.

The second direction would be to figure out just how much performance is affected by our use of Java as opposed to a native implementation. An implementation of SRSA and the forward-secure constructs in a device-native way would likely garner significant speedup. However, we would sacrifice portability, as mobile devices from different vendors almost never support the same software.

A third direction would be to investigate means of creating server-assisted signature schemes without using the generalized composition operations we employed in this paper. For example, the first forward-secure signature scheme presented by Bellare and Miner [3] did not rely on composition operators, but instead used a novel signature scheme relying on the discrete logarithm problem. An approach along these lines (i.e., not relying on composition operators) may realize improved performance in terms of space and speed.

Fourth, while the amortized cost to update a key from one time period to the next may be low, the actual cost may sometimes be quite high. For example, if the tree in figure 2 were transitioning from time period 3 to time period 4 (assuming numbering begins at 0), the entire right-hand subtree of the Product operator would need to be replaced. Ideally, the actual implementation would spread out the high cost of generating that entire tree over multiple updates; however, given that applications on mobile devices are not always on and the OS may not support any mechanism for running a process in the background (especially a Java process), this optimization may not be feasible. We intend to investigate methods for jumping from one time period to another that is several periods later without incurring the cost of the intervening updates.

Fifth, and finally, we intend to continue to experiment with different configurations of the composition operators discussed above. For example, using multiple product operators will probably result in better performance, as the effects of each will compound to provide a total key lifetime of many time periods without requiring large amounts of storage at any point in time.

7 Conclusion

In this work we have explored means for creating forward-secure server-assisted signature schemes which support key revocation. These schemes provide forward security while allowing for easy, immediate key

revocation, which makes them ideal for high-risk platforms like smart phones and PDAs. We have shown that these schemes take a significant performance hit in return for this increased security, though they remain feasible for real applications.

Our results indicate that when using composition operators to construct forward-secure signature schemes, it is best to avoid using server-assisted schemes except where necessary (i.e., where using a non-server-assisted scheme would allow a signature to be generated without consulting the server), as server-assisted schemes are much slower on phones due to the high latency of network communication.

We also have experimented with an actual implementation of these forward-secure server-assisted schemes which runs on real hardware. This experience has led us to develop several optimizations of our initial implementation, and raised new problems to solve for the next iteration of our implementation, such as the asymmetric costs associated with transitions from one time period to another.

References

1. Specification for the advanced encryption standard (AES). Federal Information Processing Standards Publication 197, 2001.
2. R. Anderson. Invited lecture. *Fourth Annual Conference on Computer and Communications Security*, 1997.
3. Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. *Lecture Notes in Computer Science*, 1666:431-448, 1999.
4. bouncycastle.org. Lightweight cryptography API for Java, 2005.
5. Jan Camenisch, Jean-Marc Piveteau, and Markus Stadler. An efficient electronic payment system protecting privacy. In *ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security*, pages 207-215, London, UK, 1994. Springer-Verlag.
6. Jan L. Camenisch and Markus A. Stadler. Efficient group signature schemes for large groups. *Lecture Notes in Computer Science*, 1294:410-424, 1997.
7. D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *CRYPTO '88: Proceedings on Advances in cryptology*, pages 319-327, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
8. D. Chaum and E. van Heyst. Group signatures. In *EUROCRYPT '91*, volume LNCS 547, pages 257-265. Springer-Verlag, 1991.
9. Eric Cronin, Sugih Jamin, Tal Malkin, and Patrick McDaniel. On the performance, feasibility, and use of forward-secure signatures. In *CCS '03: Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 131-144, New York, NY, USA, 2003. ACM Press.
10. Yvo G. Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO '89: Proceedings on Advances in Cryptology*, pages 307-315, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
11. Philip MacKenzie and Michael K. Reiter. Networked cryptographic devices resilient to capture. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 12, Washington, DC, USA, 2001. IEEE Computer Society.
12. Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Advances in Cryptology - Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 400-417, Amsterdam, The Netherlands, April 28-May 2 2002. IACR, Springer-Verlag.
13. B. Clifford Neuman and Gennady Medvinsky. Requirements for network payment: The NetCheque perspective. In *COMPCON*, pages 32-36, 1995.
14. R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, MIT, 1977.
15. M. Sirbu and J. D. Tygar. Netbill: An internet commerce system optimized for network delivered services. In *COMPCON '95: Proceedings of the 40th IEEE Computer Society International Conference*, page 20, Washington, DC, USA, 1995. IEEE Computer Society.