Symmetric Cryptography in Javascript

Emily Stark, Michael Hamburg, Dan Boneh
Computer Science Department
Stanford University
Stanford, CA
estark, mhamburg, dabo@cs.stanford.edu

Abstract—We take a systematic approach to developing a symmetric cryptography library in Javascript. We study various strategies for optimizing the code for the Javascript interpreter, and observe that traditional crypto optimization techniques do not apply when implemented in Javascript. We propose a number of optimizations that reduce both running time and code size. Our optimized library is about four times faster and 12% smaller than the fastest and smallest existing symmetric Javascript encryption libraries. On Internet Explorer 8, our library is about 11 times faster than the fastest previously existing code. In addition, we show that certain symmetric systems that are faster than AES when implemented in native x86 code, are in fact much slower than AES when implemented in Javascript. As a result, the choice of ciphers for a Javascript crypto library may be substantially different from the choice of ciphers when implementing crypto natively. Finally, we study the problem of generating strong randomness in Javascript and give extensive measurements validating our techniques.

Keywords-Javascript; cryptography; optimization;

Project homepage: http://crypto.stanford.edu/sjcl

I. INTRODUCTION

In some applications client-side encryption is needed before data is uploaded to a server cloud. Since the web browser is becoming the universal tool for interacting with remote servers, it is natural to ask whether existing browsers can perform encryption without installing additional client-side software. We found five Javascript encryption libraries [1]–[5], suggesting that there is considerable interest in encrypting data in the browser.

Websites that need client-side encryption will link to a Javascript crypto library and call the library from Javascript on the client. Clearly the site is trusted to link to correct encryption code; if the site links to bogus encryption code then encryption will provide no security at all. Nevertheless, there are several real-world settings where server-provided encryption code is adequate. First, when client-side software installation is prohibited, server-provided encryption code is the only option. Second, there are many cases where the server stores private user data, but does not want to see the data in the clear. The Mozilla Weave project [6] is a good example. Weave is a browser extension that stores all browser state (including passwords and history) in a Mozilla cloud so that the state is automatically available to the user

on any browser. Weave uses Javascript to encrypt all browser state on the client before sending it to the cloud. Here Mozilla encrypts the data to avoid the liability of having browser state in cleartext on its servers. Other examples come up in healthcare and finance where the cloud manages access to the data, but is forbidden from viewing that data in the clear. It is to the server's advantage that the data be properly encrypted by the client. In cryptographic terms the server is an "honest but curious" adversary.

Another reason for studying crypto in Javascript is its use in desktop applications. Firefox extensions, for example, are written in Javascript. Both Adobe Air and Mozilla Prism are full-fledged environments for developing desktop applications in variants of Javascript.

Javascript crypto is also used in mashups to provide secure cross-origin messaging using fragment identifiers [7]. While this mechanism is being replaced by `postMessage`, several systems still use the older fragment identifier method [7].

The discussion above suggests that a Javascript crypto library can be quite useful. Many of the existing libraries [1]–[5] are implemented for a specific task, and would not be suitable as a general-purpose library. Moreover, several implementations are not tuned to the specific characteristics of Javascript interpreters and are therefore unnecessarily slow.

Our contribution. We set out to build a general purpose symmetric crypto library in Javascript that exports a clean interface and is highly optimized. Since the Javascript library is downloaded to the browser, we want to reduce both library code size and running time. We make a number of contributions towards this goal:

- First, we observe that traditional techniques for optimizing AES, which involve hard-coding or computing various lookup tables, result in either large code size or loss of speed. All the existing Javascript implementations use one of these methods. We identified a third alternative, specific to Javascript, that keeps the code size small and improves performance.
- Second, our experiments show that certain optimizations that work well for native x86 code work poorly in Javascript. For example, a classic optimization tech-

IEEE
computer
society

nique for AES called bitslicing [8] performs very poorly in Javascript. We therefore abandoned this approach.

- Third, we found that Javascript affects the choice of ciphers. Some ciphers that are considerably faster than AES in native x86 are significantly slower than AES in Javascript. Hence, the choice of ciphers for a Javascript library can be very different from the choice for a native x86 library. We provide detailed measurements in Section V.

We hope these lessons in optimizing the code will help other Javascript crypto developers and JIT designers.

Our experiments in Section V show that in most browsers our code is at least four times faster than the fastest implementation currently available. In Internet Explorer 8 our code is **11 times faster** than the fastest current implementation. Our code is also 12% smaller than the smallest implementation currently available. Despite our performance improvements, the running times are still far worse than native code. Even in Google Chrome, the fastest browser we measured, our code runs about 46 times slower than OpenSSL's native x86 implementation.

Another difficult issue, specific to Javascript, is the question of cryptographic randomness needed for encryption. While browsers provide a `Math.random` method, it is not guaranteed to provide strong (unpredictable) randomness. In Firefox and Chrome, for example, `Math.random` is not cryptographically strong. How do we generate cryptographic randomness in Javascript? Standard methods (e.g. RFC 4086) use disk access times and scheduler non-determinism, among other things. These work poorly in Javascript since browsers are single-threaded and Javascript in the browser has no direct access to the disk. We discuss this problem and various solutions in detail in Section IV. One option is to rely on the user to provide randomness via mouse movements or keyboard input. We present in Section IV the results of a user study that measures the amount of entropy generated by tracking mouse movements as users interact with various web pages. This entropy collection method is transparent to the user.

We note that Google recently introduced Native Client (NaCl) technology [9], which enables a remote site to send x86 code to the browser and have the code execute on the CPU. In many years, once NaCl is deployed universally, there will be less need for Javascript crypto. Until then, Javascript crypto will continue to be of interest.

II. Background

Symmetric cryptography is primarily used for data integrity and confidentiality. Other applications, such as user authentication, can be built from these primitives. Therefore, we focus on building primitives that provide integrity and confidentiality with integrity. We made the following design decisions early on:

- We experimented with data integrity using HMAC-SHA-256 and discovered that SHA-256 is considerably slower than AES in Javascript. This holds across all mainstream browsers (see Section V). Consequently, we abandoned SHA-256 for integrity and instead focused only on AES-based integrity methods such as AES-CMAC [10]. It is worth noting that for native x86 code (OpenSSL), SHA-256 is about 10% slower than AES-128, but in Javascript (Chrome) it is about 5 times slower than AES-128.

- We also implemented modes that provide both confidentiality and integrity, specifically the OCB [11] and CCM [12] modes. These modes take as input a key, a message and a nonce. The nonce must never be reused for a given key. Once the nonce is specified, encryption is deterministic. In some applications, the developer using the library will want to specify the nonce (for example, if the nonce is a packet counter). In others, the developer will want to ignore the nonce and have the library choose it at random for every message. We support both methods by using our Javascript randomness library to choose a random nonce when needed.

III. Fast and Small AES in Javascript

The AES block cipher [13] consists of a sequence of rounds. In each round one evaluates mixing functions and a 256-byte lookup table called an S-box. The S-box itself can be computed using very little code.

A. Maximizing precomputation

Implementations of AES typically choose one of two techniques for evaluating the mixing functions and the S-box. The first approach is to compute the mixing function and S-box as needed during encryption. This keeps the code small, but increases the running time. The second approach is to precompute both the S-box and mixing functions and store the precomputed tables in the code. This speeds up encryption and decryption times, but increases the code size. For example, the OpenSSL implementation contains some 8KB of hard-coded tables. This is negligible for local use, but would fill multiple packets over a typical network. Furthermore, when embedded in C code, the tables take up 24KB in source form, and even after compression they are over 10KB.

Existing Javascript implementations avoid storing lookup tables for the mixing function so as not to increase the library size. Instead they only hardcode the 256-entry S-box, and sometimes log tables for finite-field multiplication. Omitting the round transformation tables comes at a cost to performance, since round transformations must be computed rather than looked up. Hardcoding the round transformation tables would add significantly to code size.

Our approach. In our Javascript implementation we propose a different approach that keeps the code small and speeds up encryption/decryption. Instead of embedding tables in the code, our library contains code that precomputes all tables on the browser before encryption or decryption begins. Since code to precompute the tables is much smaller than the tables themselves, we keep the library size small. We reap the benefits of precomputed tables by quickly building the tables on the browser before the first encryption. The only values we hardcode are the 10 round constants, which are 28 bytes total.

Precomputing the tables takes about 0.8ms on Google Chrome and about 8ms on Internet Explorer 8. Because this saves several KB of code, it is nearly as fast to precompute the AES tables as it would be to load them from disk. Moreover, computing the tables in the browser is far faster than downloading them over the network.

When encrypting a short message, say a single AES block, it may seem that precomputing the AES tables is a waste and that we should instead compute the tables on the fly as needed. We argue that, for three reasons, precomputation is always a good idea, regardless of the message length. First, using different code for short messages would increase the code size. Second, the structure of the S-box makes it so that precomputing the entire S-box at once is faster than computing each entry individually. Because encrypting a single block with a 128-bit key requires 160 S-box evaluations, precomputing the entire S-box is faster than encrypting a single block without it. Third, computing the S-box on the fly exposes us unnecessarily to a timing attack.

Our library always precomputes the AES tables the first time `encrypt` is called.

B. Loop unrolling

While developing our AES implementation, we considered the extent to which we should unroll loops. In C implementations of AES, such as OpenSSL [14], there is little reason not to unroll loops, but in Javascript, unrolling loops results in an undesirable increase in code size. We experimented with three levels of unrolling in our core AES implementation:

1) Leaving all loops rolled, the technique used by most Javascript AES implementations.
2) Unrolling short loops (four or fewer iterations), in hopes of increasing performance at a low cost to code size.
3) Unrolling the main round transformation loops (10-14 iterations per block), the technique used by OpenSSL.

In our experiments, unrolling short loops had little effect on performance. Unrolling the round function increased speed significantly on some browsers but at the cost of a 55% increase in code size. We also noted that the performance increase was most notable in browsers with older Javascript interpreters; unrolling sped up the implementation by 50-75% in Internet Explorer 8 and Firefox 3.0, but only by roughly 20% in Chrome and actually slowed down encryption in Safari 4 and Firefox 3.5. As Javascript interpreters advance to JIT compilation, the slower but smaller implementation with no unrolled loops will be the better choice.

Unrolling short loops in our OCB-mode code increased speed by roughly 5% in Chrome and 12% in Firefox 3.5 at the cost of a 12% increase in code size.

C. Bitslicing

Bitslicing is an optimization technique where encryptions are broken down into logical bit operations that are then computed in parallel. Bitslicing can significantly increase performance of symmetric ciphers, particularly AES [8], [15], [16]. In particular, [16] claims to be the fastest AES implementation to date.

Bitslicing is ineffective in Javascript. It decreases performance in most browsers for two reasons. First and foremost, bitslicing works best when we have access to a wide words, as in 128-bit SIMD in native x86. In Javascript we only have access to 32-bit-wide operations, making bitslicing far less effective. Second, Javascript JITs seem to perform poorly with large numbers of local variables, which would be hard to avoid.

The bitsliced implementation proposed in [8] uses 132 gates for the SubBytes function alone, which corresponds to 16 S-box table lookups. The times for bitwise operations and table lookups can be found in Table I, which suggests that bitslicing (132 bitwise operations) would be slower than a conventional AES implementation (16 table lookups) in all browsers except Firefox 3.5 beta, even if register spilling is not an issue.

D. Unifying encryption and decryption functions

In another effort to reduce code size, we combined the encryption and decryption functions of the AES core by using different lookup tables depending on whether or not the programmer set a `decrypt` flag. For our standard implementation, this decreased the code size by 14%.

IV. RANDOMNESS IN JAVASCRIPT

All cryptography libraries need a strong random number generator, both to guarantee that nonces do not repeat and to enable the user to generate random challenges, secret keys, etc. If any high-quality source of randomness is available, we can extend it indefinitely using a cryptographic pseudo-random number generator (PRNG). We considered several sources of entropy; each has advantages and disadvantages.

We need two types of entropy for cryptography. To generate secret keys, we require that the attacker cannot predict the entropy pool. To generate nonces, we only require that the PRNG's output not repeat, so it will suffice that the attacker cannot control the entropy pool. Because the

Table I
TIMINGS FOR BITWISE OPERATIONS AND TABLE LOOKUPS ACROSS BROWSERS.

	Time (ns)				
	Chrome	IE 8b	Safari 4	Firefox 3.0	Firefox 3.5b5
Fastest bitwise operation	1.76	35.4	10.1	4.96	1.55
Table lookup	1.81	32.9	10.7	4.48	23.80
132 bitwise operations	232	4673	1333	655	205
16 table lookups	29	526	172	72	381

attacker can predict more measurements than it can control, the random number generator can provide nonces more quickly than it can provide secret keys.

A. Explicit random number generators

Javascript exposes a random number generator as `Math.random`. Unfortunately, except in Safari on Windows (and Safari on Mac after revision 39510), this function is not cryptographically strong, and determining its strength would rely on potentially brittle browser version detection code in Javascript. Firefox and Internet Explorer use linear congruential generators, and so does Chrome (even though Chrome is based on WebKit, it does not use WebKit's Javascript implementation). Of course, it is still useful to fold data from `Math.random` into the seed of a PRNG. Its state may not be accessible to an attacker, and on some few browsers it is cryptographically strong. Since folding weak entropy into the state of a PRNG is harmless, this step will have no impact on browsers that implement a non-cryptographic `Math.random`.

Even when `Math.random` is weak, it would be acceptable for generating nonces if its state were large enough, as an attacker would not be able to force a repeat with meaningful probability. However, we found that several browsers with a weak PRNG have state space smaller than 48 bits. In this case, an attacker could cause a repeat by calling `Math.random` until its output is divisible by, say, 2^{24}, leaving only 2^{24} possible nonces and causing a repeat after an expected 2^{12} operations.

Consequently `Math.random` cannot be trusted as the only source of entropy for our library. An alternative is to use the dedicated cryptographic PRNG `window.crypto.random`, which first appeared in Netscape 4. If present, this function would supply all the entropy we need. Unfortunately, as of this writing, `window.crypto.random` is not implemented in any major browser. We therefore examine stand-in methods to generate entropy.

B. Operations which implicitly use entropy

All major web browsers support SSL, which means that they have cryptographic PRNGs for internal use. Furthermore, some browsers implement Javascript-accessible cryptography routines for client-side certificate generation and signing. For example, Firefox has `crypto.generateCRFMrequest`, which creates a new random private key and corresponding certificate signing request.

It is tempting to extract entropy by requesting that the browser perform some randomized cryptographic operation, and then hashing the result. However, this has undesirable side effects. Calling `generateCRFMrequest` uses a non-negligible amount of CPU time and briefly pops up a dialog box informing that the operation "may take several minutes". More seriously, when the operation completes, NSS (the Firefox SSL library) stores the private key in a hidden database. Therefore every call to `generateCRFMrequest` irreversibly expands this database. These side effects make the strategy untenable.

C. RFC 4086

RFC 4086 recommends various strategies for generating random numbers. Unfortunately, it does not apply particularly well to the browser. Most of the hardware (disk drives, sound cards, etc) which is conventionally used to generate randomness is not accessible from Javascript.

However, a small amount of entropy can be extracted from network latencies. If we attach `load` event handlers to objects in the page which have not yet loaded then the loading time should give some entropy. When the object is in disk cache, it will load in a few milliseconds; this may allow us to extract entropy from disk timings. When it is in memory cache, it will load nearly instantaneously. By discarding load times less than 2 milliseconds, we avoid treating these loading times as random.

This approach to collecting entropy is not secure against a network attacker or against another user on the local machine since they may have access to these load times. Hence, these measurements can be folded into the PRNG state, but cannot be trusted as the sole source of entropy for the PRNG seed.

D. User interaction

Our main entropy source is user interaction, specifically mouse movement. Our PRNG's initialization code attaches an `onmousemove` event handler to the `window`, which gives us the x and y coordinates of the mouse whenever it moves. This source is secure against a network attacker, and seems likely to be secure against a web attacker as well,

though timing information may leak some small amount of entropy. But it is not obvious how much entropy mouse movements on a web page actually have.

In some cases the browser may send `mousemove` events even though the mouse did not move. We accord no entropy to such events, though we still add them to the pool. Similarly, we accord no entropy to events that show the mouse moving or accelerating uniformly.

User study. To determine the amount of entropy, we measured users' mouse movements on three different websites: a forum, a survey and a blog.

- on the forum site, users read and post comments;
- the survey site asks users to answer a number of questions;
- on the blog users mostly read posts by others.

We collected over 4,000,000 samples from over 9,000 different users. We attached an `onmousemove` event handler, which calls into an event handler whenever the mouse moves within the window, giving the x and y coordinates that it moved to. We also recorded the time in milliseconds when the event fired.

To estimate the amount of entropy in the samples, we tried to predict the x, y and t coordinates for each sample from previous samples using three models – constant, linear and quadratic. For the constant model, we estimated that the position of the current sample would be the same as the previous one. For the linear model, we estimated that the difference in position would be the same as for the previous pair of samples. For the quadratic model we did the same with differences of differences. We predicted that t would always be linear. We removed the samples which were predicted perfectly, just as our entropy collector does. For each sample, we chose the model that gave the lowest error in the prediction. We then computed the Shannon entropy of the resulting distribution. Results are shown in Table II.

On all three sites the mean is over 5.5 bits of entropy per mouse move event. Along with the mean, we show in Table II the minimum and lowest 5th percentile of users. We show the lowest 5th percentile in order to give some idea as to how rare the minimum case is. The minimum and 5th percentile are shown from users with at least 32 samples in order to prevent users with very few samples from skewing the minimum.

We conservatively estimated each site to have at least 5.5 bits per sample on average, but because of the outlying minimal cases, we chose to assign only 2 bits of estimated entropy to each sample. Since we require 256 bits of estimated entropy before we consider the generator to be seeded, seeding should be secure even if this estimate is too high by a factor of 2 to 3.

Both user interaction and timing measurements are event-driven entropy sources and cannot provide entropy on demand. If the `encrypt` function is called before the

Table II
MOUSE MOVEMENT ENTROPY IN BITS PER SAMPLE

Site	Users	Samples	Mean	Min	5%
Forum	9245	4037k	5.5	2.5	4.2
Survey	38	58k	7.6	5.2	5.2
Blog	255	141k	6.6	2.2	4.6

generator is seeded there is no choice but to return an error. This is because Javascript is single-threaded and the PRNG cannot block to wait until it is seeded. If the calling code happens to call `encrypt` before the generator is seeded, it needs to handle the resulting error by asking the user to move his or her mouse until the generator is seeded (which is indicated by a callback).

Figure 1 shows how long it takes before the generator is seeded by regular user interaction with the page. Data is provided for the three sites we tested. Since we extract 2 bits of entropy per mouse move sample, we need 128 samples to generate a 256-bit AES key. **Our data show that the median time for the generator to be seeded to the default level of 128 samples is 9, 28 and 41 seconds on the survey, forum and blog, respectively.** These measurements are consistent with our intuition:

- On the survey site users move their mouse from field to field as they answer questions, resulting in fast seeding of the generator.
- On the blog site users mostly read and hardly move their mouse, causing the generator to take some time before it is seeded.

Halprin and Naor [17] proposed an alternate approach for generating cryptographic randomness from user interaction by engaging the user in a game. While not transparent to the user, their approach is another viable method for generating cryptographic randomness with the user's help.

E. Cookies

Server-side authentication cookies often have considerable entropy. We provide an option to fold cookies into the entropy pool. To avoid leaking the cookies if the PRNG state is compromised, we still require additional entropy from user interaction. We can also encode our PRNG state into a cookie or into local storage in order to save entropy between page loads. Neither of these methods is particularly trustworthy, as an attacker may be able to see or overwrite our cookies, but they provide defense in depth.

F. PRNG

Our PRNG itself is a modified version of the Fortuna PRNG [18]. Fortuna produces pseudorandom words by running a block cipher in counter mode. The counter always increments and is never reset, which prevents short cycles. After every request for pseudorandom words is satisfied the
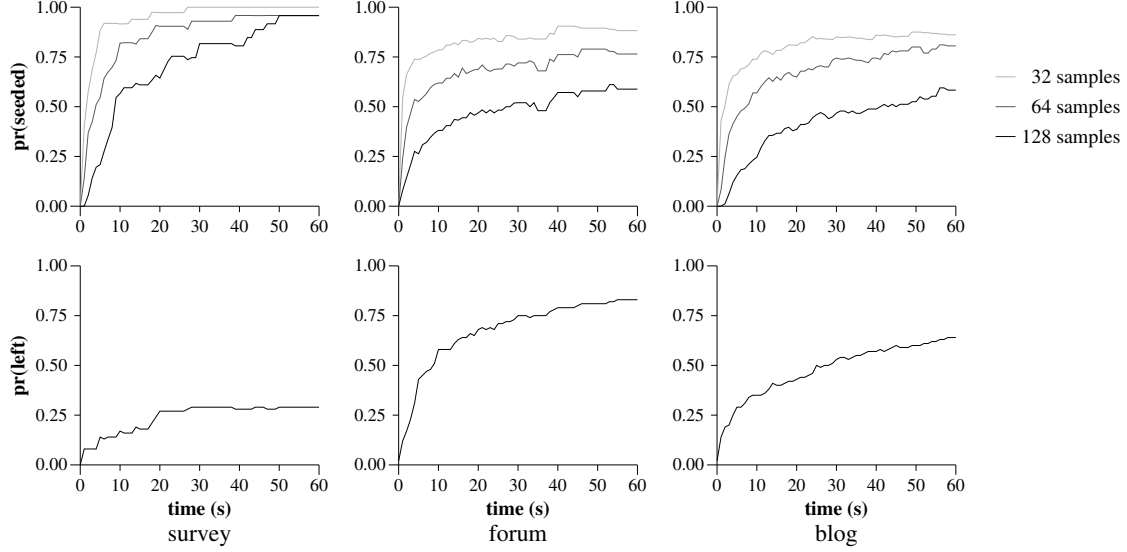
Figure 1. Time required to seed the PRNG from regular user interaction with the page. The top three graphs show the fraction of users still on the page who are seeded to each strength. The bottom three graphs show the fraction of users who have left the page.

key is replaced by the next output of the generator. This "rekey" step prevents an adversary who compromises the generator from recovering previously-generated data.

Additionally, the generator reseeds itself using entropy collected in n "pools" numbered 0 through $n-1$. The pools are implemented as streaming SHA-256 contexts, a common heuristic for entropy extraction. Collected entropy is divided by round-robin between the pools. On the ith reseed, the generator uses pools 0 through m for the largest m such that 2^m divides i. As a result, the mth pool is only used every 2^m reseeds, so that even if an adversary can predict or control all but an ϵ fraction of the seed data, the generator will still recover from a compromise after $O(-\log \epsilon / \epsilon)$ samples, a factor of $-\log \epsilon$ slower than the fastest possible recovery.

The Fortuna design uses $n = 32$ pools, but the round-robin makes the initial seeding very slow. This doesn't matter in the context of a system-wide PRNG which saves its state between reboots, but on the Web, it matters. As a result, we instead start with $n = 1$, and create a new pool every time the last pool is used. This doesn't change the $O(-\log \epsilon / \epsilon)$ recovery bound.

Unlike Fortuna, we include a rudimentary entropy estimator so that we can estimate when the generator is safe to use. By default, we require 256 bits of estimated entropy before we consider the generator to be seeded. After the first seeding, our generator reseeds itself every time pool 0 has collected 80 bits of estimated entropy, but no sooner than 100 milliseconds after the last reseed. To save computation, we defer reseeds until the next time the generator is called.

V. EXPERIMENTS

Our measurements show the effect of various implementation choices on the library's performance and code size. We also compare our library to previous implementations of AES.

A. Comparison to other implementations

We compared our implementation with five existing AES implementations in Javascript:

- Clipperz, an online password manager [1]
- An implementation by Eastern Kentucky University's Eugene Styere [3]
- Google's Browser Sync [2]
- Javascrypt, a cryptography tool by Fourmilab's John Walker [4]
- An implementation by Movable Type's Chris Veness [5]

For each of these implementations, we "crunched" the code with an online Javascript minifier [19] to remove comments and long variable names, then measured the code size. We measured encryption speed on five different browsers: Google Chrome, Microsoft Internet Explorer 8 beta, Apple Safari 4, and Mozilla Firefox 3.0 and 3.5 beta 5. We enabled the Javascript JIT compiler in Firefox 3.5. Safari 4 and Google Chrome also use JIT compilers. Measurements were taken on a Lenovo Thinkpad T61 with an Intel Core 2 Duo processor (2.1 GHz) running 32-bit Windows Vista.

Table III shows code size and running times, in addition to the speed improvement over the fastest previously existing implementation for each browser. The improvement in speed is due to our construction of large lookup tables in browser memory that allow us to compute mixing functions quickly.

Table III
CRUNCHED SIZES AND SPEEDS FOR VARIOUS JAVASCRIPT AES IMPLEMENTATIONS.

Implementation	Size (B)	Speed (kB/s)				
		Chrome	IE 8b	Safari 4	Firefox 3.0	Firefox 3.5b5
Our code	5687	585.2	60.4	264.8	97.4	451.6
Clipperz	9395	58.6	2.1	12.3	4.8	5.4
EKU	14667	99.0	2.7	136.9	5.1	42.8
BrowserSync	15916	125.9	5.2	231.5	21.8	62.3
Javascrypt	6455	16.2	1.3	33.6	13.3	13.9
Movable Type	6460	111.4	5.2	110.7	13.1	45.8
Improvement	12%	365%	1062%	14%	347%	625%

The improvement in code size is due to precomputation: the code to precompute our lookup tables is about 60% smaller than a hard-coded S-box used in previous implementations.

B. Effects of precomputation

Precomputing both the S-box and the round transformations tables T results in a worst-case precomputation time of fewer than 10 milliseconds each time the library is loaded. Hard-coded tables speed up precomputation at the cost of increased code size. In some browsers, precomputing the tables may be even faster than loading them from disk.

Table IV shows that hardcoding nothing and instead precomputing all tables introduces minimal performance degradation, but greatly reduces code size. Hardcoding both the S-box and the T-tables can remove at most eight milliseconds from the time taken to load the library, but that hardly seems worth the code bloat.

Table IV shows that our approach to precomputation provides the speed benefits of hardcoding AES tables in the code, but without the code bloat.

C. Effects of loop unrolling

Table V shows performance gains from unrolling loops in the core encrypt function. The first implementation is our standard implementation with no unrolling. The second implementation unrolls short loops with 4 or fewer iterations. The third implementation unrolls the round transformation loop as well, which iterates 10 times per block for 128-bit keys. This is the technique used by OpenSSL [14]. As Table V shows, unrolling short loops achieves no significant performance improvement. Unrolling the round function, however, shows significant improvements in some browsers at the cost of increased code size. In some of the newer Javascript interpreters, loop unrolling shows less of a performance increase, suggesting that loop unrolling in Javascript is currently effective but will become less relevant as users adopt more modern interpreters.

We also experimented with loop unrolling in the CCM and OCB implementations. These modes of operation wrap the core AES code. There was no significant improvement in CCM and OCB performance in Chrome. Firefox 3.5 was improved by approximately 5% and 12%, respectively, by unrolling short loops (mostly four or fewer iterations). This same optimization had little effect on the AES core; however, there are more short loops in the OCB implementation than AES, which accounts for the performance improvement in OCB. Still, this small performance increase came at the cost of a 12% increase in code size, motivating our decision to use an implementation of OCB with no unrolled loops in our library. Results for the standard and unrolled implementations of OCB are shown in Table VI.

D. Comparison to other algorithms

To ensure that AES is a good choice for a Javascript crypto library, we compared our optimized implementation to our own implementations of other algorithms. Table VII shows the results of these comparisons. We found that SHA-256 is slower across all browsers, which motivated our decision to abandon HMAC-SHA-256 for integrity and instead use AES-CMAC.

We compared Javascript AES with a Javascript implementation of Salsa20/12 [20], one of the fastest eSTREAM ciphers. A native x86 implementation of Salsa20/12 is about 5 times faster than a native implementation of 128-bit AES. Surprisingly, Table VII shows that when both algorithms are implemented in Javascript, Salsa 20/12 is comparable in speed to AES. We believe that this discrepancy is primarily due to Javascript's lack of 128-bit SIMD instructions or of 64-bit registers, and secondarily due to Salsa20/12's larger state spilling to memory.

VI. CONCLUSION

We described an optimized Javascript implementation of symmetric cryptography. Our primary contribution is an approach which differs from typical AES implementations and allows us to reduce code size and increase speed. Instead of doing all the cipher computations at run-time or including large lookup tables in code, our implementation precomputes tables as soon as the first cipher object is created. This precomputation is fast enough to be a win when encrypting or decrypting messages of any length, and it is much smaller than hardcoding the tables themselves.

We studied various optimization tradeoffs for AES and modes of operation (OCB and CCM), and we showed that

Table IV

CRUNCHED SIZES AND SPEEDS FOR OUR AES IMPLEMENTATIONS WITH VARIOUS AMOUNTS OF PRECOMPUTATION.

Hardcode	Size (B)	Precomputation time (ms)				
		Chrome	IE 8b	Safari 4	Firefox 3.0	Firefox 3.5b5
Nothing	5687	0.89	7.85	1.59	2.93	3.03
S-box only	8788	0.78	7.33	1.59	2.77	3.04
Everything	32799					

Table V

CRUNCHED SIZES AND SPEEDS FOR OUR AES IMPLEMENTATIONS WITH VARIOUS AMOUNTS OF LOOP UNROLLING.

Unroll	Size (B)	Speed (kB/s)				
		Chrome	IE 8b	Safari 4	Firefox 3.0	Firefox 3.5b5
None	5687	1524	153	256	260	326
Short	6485 (+14%)	1596 (+5%)	136 (−11%)	248 (−3%)	253 (−3%)	313 (−4%)
Round	8814 (+55%)	1836 (+20%)	233 (+52%)	223 (−13%)	453 (+74%)	270 (−17%)

Table VI

CRUNCHED SIZES AND SPEEDS FOR OUR OCB IMPLEMENTATION WITH AND WITHOUT LOOP UNROLLING.

Unroll	Size (B)	Speed (kB/s)				
		Chrome	IE 8b	Safari 4	Firefox 3.0	Firefox 3.5b5
None	3772	183	34.9	54.2	60.0	42
Short	4221 (+12%)	193 (+5%)	34.7 (−0%)	53.9 (−1%)	59.8 (−0%)	47 (+12%)

Table VII

PERFORMANCE OF AES, SHA-256, AND SALSA 20/12 (OUR IMPLEMENTATIONS).

Algorithm	Speed (kB/s)				
	Chrome	IE 8b	Safari 4	Firefox 3.0	Firefox 3.5b5
AES	679.4	63.2	248.0	124.5	347.2
AES unrolled	**753.6**	**92.0**	**214.0**	**173.5**	**289.4**
SHA-256	131.9	38.6	71.3	50.9	153.2
Salsa20/12	452.2	148.8	266.0	161.8	283.1

certain standard optimizations are ineffective in the browser. Our data suggests that SHA-256 performs far worse than AES and consequently Javascript integrity schemes based on AES are a better choice than integrity schemes based on HMAC-SHA-256. Similarly, certain ciphers, like Salsa 20/12, consistently outperform AES in native code but do not in Javascript. Our library, which is publicly available on the Internet [21], is four times faster and 12% smaller than previous Javascript AES implementations.

We also discussed the issue of generating cryptographic randomness in the browser. We hope that future browsers will provide a clean solution to this issue, for example by implementing Netscape's `window.crypto.random`.

REFERENCES

[1] Marco and G. Cesare, "Clipperz online password manager," 2007, http://www.clipperz.com.

[2] "Google Browser Sync," 2008, http://www.google.com/tools/firefox/browsersync/.

[3] E. Styere, "Javascript AES Example," October 2006, http://people.eku.edu/styere/Encrypt/JS-AES.html.

[4] J. Walker, "JavaScrypt: Browser-Based Cryptography Tools," December 2005, http://www.fourmilab.ch/javascrypt.

[5] "Javascript implementation of AES in counter mode," http://www.movable-type.co.uk/scripts/aes.html.

[6] M. labs, "The weave project," 2007, https://wiki.mozilla.org/Labs/Weave/0.2.

[7] C. Jackson, A. Barth, and J. Mitchell, "Securing frame communication in browsers," in *Proc. of USENIX Security 2008*, 2008.

[8] D. S. Chester Rebeiro and A. Devi, "Bitslice Implementation of AES," in *Lecture Notes in Computer Science*. Springer Berlin, 2006, pp. 203–212.

[9] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: a sandbox for portable, untrusted, x86 native code," in *Proceedings of IEEE Security and Privacy*, 2009.

[10] NIST, "Special publication 800-38b, recommendation for block cipher modes of operation: The cmac mode for authentication," 2005.

[11] T. Korvetz and P. Rogaway, "The ocb authenticated-encryption algorithm," Internet Draft draft-krovetz-ocb-00.txt, 2005.

[12] NIST, "Special publication 800-38c, recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality," 2004.

[13] J. Daemen and V. Rijmen, "The Rijndael Block Cipher," AES Proposal, March 1999, http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[14] "OpenSSL: The Open Source toolkit for SSL/TLS," http://www.openssl.org.

[15] M. Matsui and J. Nakajima, "On the power of bitslice implementation on intel core2 processor," in *Cryptographic Hardware and Embedded Systems*, 2007.

[16] E. Kasper and P. Schwabe, "Faster and timing-attack resistant aes-gcm," Cryptology ePrint Archive: Report 2009/129, 2009.

[17] R. Halprin and M. Naor, "Games for extracting randomness," in *Proceedings of SOUPS '09*, 2009.

[18] N. Ferguson and B. Schneier, *Practical Cryptography*. Wiley Publishing, Inc., 2003.

[19] D. Edwards, "A javascript compressor," 2007, http://dean.edwards.name/packer.

[20] D. Bernstein, "Salsa20 specification," eSTREAM Project algorithm description, http://www.ecrypt.eu.org/stream/salsa20pf.html.

[21] Emily Stark, Mike Hamburg, and Dan Boneh, "jsCrypto," 2009, http://crypto.stanford.edu/sjcl.