Batteries Included

Features and Modes for Next Generation Hash Functions

Stefan Lucks[1], David McGrew[2], Doug Whiting[3]

[1]Bauhaus-Universität Weimar, Germany, [2]Cisco Systems, USA,
[3]Exar Corporation, USA

Abstract. The first generation of dedicated hash functions, starting with MD4 and including SHA-1 and the SHA-2 family, just defined plain hash functions. As it turned out, hash functions were employed for many applications the original hash function designers had not anticipated, and users thus defined their own modes of operation to satisfy their needs. Today's designers and decision makers have the chance to learn from the past: they know already about the plethora of use cases for hash functions. Future hash standards should accommodate these use cases by selecting designs that can address those use cases securely and efficiently.

1 Introduction

Regardless of which finalist will be chosen, the new SHA-3 standard will be more than just another standardized hash function: It will be the first standardized member of a new generation of hash functions. What benefits will it provide, beyond not being broken? (Note that the SHA-2 family of first generation hash functions isn't broken either!)

The first generation of dedicated hash function standards – either defacto standards such as MD5 or "official" standards such as SHA-1 and the SHA-2 family – shared

1. a common design principle [8, 15],
2. a common set of weaknesses, especially the famous length extension property,
3. a common usage model (a message of arbitrary size is read sequentially; eventually an n-bit hash value is emitted, where n is the single output size supported by the hash function), and
4. a common – and fairly simple – view about the security requirements for the hash function, strictly determined by the output size n (i.e., collision attacks at less than about $2^{n/2}$ units of time are fatal, and preimage attacks at less than about 2^n units of time are fatal).

Also common is the approach to model the hash function as a random oracle, though this requires a workaround to defend against the length extension weakness. Modeling hash functions as random oracles has initially been handled rather informally [2]. Beginning with [7], researchers followed a more formal approach by studying the *indifferentiability* of a hash functions from a random oracle, assuming some compression functions or block ciphers used inside the hash function work like their ideal counterparts. A carefully designed hash function can be indifferentiable if an attacker is restricted to $\ll 2^{n^*/2}$ oracle queries. Here, n^* is the internal state size (or the size of the "chaining variable") of the hash function.[1] But actually, many other non-standard assumptions on the security of a hash function were made when it seemed to fit to the application or mode.

As we pointed out above, designers of first generation hash functions used the size of the output as the indicator to model the security of hash functions. Of course, the size of a hash value bounds the workload for a preimage or a collision attack from above. E.g., if the size of a hash value is n bit, collisions can be found in time proportional to $2^{n/2}$. As we will elaborate below, other security properties for hash functions are not affected immediately by changing the size of a hash value, and applications may need to increase or decrease the hash value size while maintaining a constant security level with respect to certain attacks.

In spite of their shortcomings, the hash functions from the first generation had been extremely successful. In a recent study, citations of SHA-1 outnumbered those of AES in Internet standards by 3:1 [10]. These functions have been adopted for a wide range of applications not foreseen by the original hash function designers. These applications include, e.g.,

- keyed hashing (i.e., to use the hash function in some keyed way to compute a message authentication code, MAC (see [1], Section 9.5.2 from [14], and references therein),
- processing the plaintext in public-key encryption schemes, (e.g., in the context of the optimal asymmetric encryption padding, OAEP [3, 9]),
- key derivation,
- entropy extraction from non-uniform random sources,
- etc.

Users somehow adapted the hash function at hand to make it work for their application. This implied different (and thus incompatible) ad-hoc

[1] This is an example of a modern security requirement for hash functions, where the concrete security can be independent from the output size n.

ways to achieve the same thing, and sometimes it even introduced subtle security flaws. A second-generation hash function such as the coming SHA-3 should not need ad-hoc solutions for common usage patterns.

Specifically, the next generation of hash functions should provide standardized support

- for message authentication, based on the hash function,
- for the generation of "nonstandard" output sizes,
- for the definition of secure "personalized" hash functions,
- for parallel hashing (i.e., some "tree hashing" mode),
- and for hash-based signatures without relying on primitives/assumptions from public-key cryptography.

Below, we will explain what exactly we mean by supporting the above requirements and why we think they are important. We stress that not all these issues need to be dealt with the SHA-3 standard itself. Most of them can alternatively be supported by sibling standards that consider *hash modes of operation*. Several existing standards can be considered to be hash modes, including conventional hashing for digital signature applications (FIPS 186-3), randomized hashing for signatures (SP 800-106), hash-based message authentication (SP 800-198), hash-based key derivation (SP 800-108, SP 800-56A), and entropy extraction using hash functions (SP 800-90A). The SHA-3 standard should not try to directly address all of these requirements. However, that standard should aim to select a function that offers potentially advantageous ways of meeting these requirements. Follow-on sibling standards can then realize that potential, and can focus on the important details of each requirement. **What is essential to the SHA-3 process is the fact that the choices made now limit our options for realizing these potential benefits.**

Below we outline generic ways to support these requirements. That doesn't mean they should necessarily be implemented by such a generic technique – depending on the details of the candidate that will eventually be chosen as SHA-3, there may be better ways to realize these features. **The current SHA-3 conference is the ideal point of time to discuss these requirements and how to satisfy them.** Otherwise, once SHA-3 has been fixed, users will just define their ways to use SHA-3 to satisfy their needs.

Remark: The first and third author have been involved in the design of the Skein hash function. Their expectations for the next generation of hash functions have determined many design decisions for Skein, which thus addresses all the above needs. However, this paper isn't meant as an advertisement for Skein – in fact, there are generic ways to implement any of these features, based on any of the SHA-3 finalists. The point we are trying to make is that these need to be addressed *now*, to avoid a proliferation of different incompatible and sometimes insecure ad-hoc solutions.

Outline of the paper: Below, we will describe the features or modes that we believe SHA-3 should provide. We will categorize our features into "essential" ones (see Section 2), non-essential but "important" ones (Section 3) and other "nice to have" features (Section 4). Section 5 discusses interoperability issues when different features / modes are used simultaneously, Section 6 highlights the importance of security arguments and explicit assumptions, and Section 7 concludes the paper.

2 Essential Features

2.1 Support for Hash Function Based Message Authentication

Almost immediately after the initial publication of the first generation of hash functions, people started using them for message authentication, i.e., for keyed instead of unkeyed hashing. Note that keyed hashing is the second most common cryptographic algorithm cited in Internet standards, with HMAC outnumbering AES by 2:1 [10]. Soon people discovered that obvious solutions, such as using the Key K as a prefix for the message M (i.e., $MAC_K(M) = H(K||M)$) was not as secure as expected, regardless of which of the first generation hash function they where using. The reason for the weakness is the Merkle-Damgaard design principle, and, mainly, the length extension weakness. Eventually, people came up with a well-analyzed and provably secure workaround: The HMAC-construction [1] is a generic construction which became a de-facto standard for hash-based message authentication:

$$\text{HMAC}(K, M) = H((K \oplus \text{opad})||H((K \oplus \text{ipad})||M))$$

(with ipad $\neq$ opad being public constants).

None of the SHA-3 finalists suffers from the length extension weakness, so the HMAC construction – as important as it has been for the first generation hash functions – is entirely unnecessary, whichever finalist is chosen. (Though HMAC would be secure with any SHA-3 finalist, as far as we can tell.) Message authentication with HMAC is a workaround needed by first generation hash functions, but it is unneeded and undesirable when using SHA-3. Having a standard message authentication mode incorporated into SHA-3 would be simpler, and more efficient, than using a separate HMAC function in conjunction with that hash. HMAC invokes the underlying hash function twice, always making at least two calls to the internal compression function for a single message.

HMAC is less efficient on short messages, because of its per-invocation overhead. As an example, one application where this inefficiency is pronounced is the Secure Real-time Transport Protocol (SRTP), which uses HMAC-SHA1 to protect audio packet streams that can have payloads as short as ten bytes. Table 2.1 illustrates this, showing how throughput is considerably higher for longer packets. (This test was performed on a 2.5Ghz Intel Core2 Duo using an adaptation of the SRTP reference implementation, which uses portable C code and caches HMAC state after the processing of the IPAD variable.)

Codec	Authenticated data (bytes)	Megabits/second
G.729 (10)	22	118
G.729 (20)	32	171
G.726-32	52	270
G.711 (80)	92	378
G.711 (160)	172	561
Wideband	332	703
Wideband	652	845

Table 1. HMAC-SHA-1 throughput as a function of authenticated data size, for data sizes that correspond to the use of common audio codecs in SRTP.

SHA-3 should provide an new standard message authentication mode, rather than rely on HMAC. To authenticate many short messages under the same key, a single compression function call per message suffices for SHA-3.

2.2 Provide for "nonstandard" output sizes

Often, the output length n provided by a hash function H doesn't quite fit the size s which the application provides for the hash output. We distinguish two cases: $s < n$, i.e., the native n-bit hash output is too long, and $s > n$, i.e., the native output is too short.

As we will argue below, it is easy to enhance a given hash function H, originally developed to support a fixed n-bit output, and turn it into a hash function H_s to support s bits of output. But then, even for exactly n bits of output, H_n has to be used, not H itself. For this reason, we consider it essential that this issue is solved before H is established as the standard for n output bits.

Case 1: The native hash output is too long

If $s < n$, the solution appears simple: just define a new s-bit hash function H' by truncating $n - s$ bits away:

$$H'(M) = \text{truncate}_s(H(M)).$$

Most of the time, this seems to work well in practice. **What about the security of H', compared to the security of H?** The traditional security reasoning from the first generation of hash functions seems to indicate that H', with less output bits than H, will be less secure than H. Indeed, preimage attacks on H' can be performed in 2^s units of time. Similarly, collision attacks can be performed in $2^{s/2}$ units.

But resistance against many other attacks remains unaffected by the transformation of H into H'. E.g., if the internal state size of H is n, internal collisions need no more than $2^{n/2}$ – the same for both H and H'. Assuming H is well-designed, Joux-like multicollison attacks [11] against H and H' should take the same $2^{n/2}$ units of operation in spite of the smaller output size of H' [13].

Furthermore, one technique to improve first-generation hash functions with respect to their ability to imitate a random oracle is to truncate away a significant number of output bits. Let H be a first-generation hash function and model the compression function of H as an ideal one. By the length extension property, H is easily differentiable from a random oracle. On the other hand, if s is sufficiently small, then H' is indifferentiable from a random oracle and, in this sense, more secure than H [7].

There is one issue, though. The fact that there are two different hash functions H and H' which are so closely related that given $H(M)$ for a secret M one immediately knows $H'(M)$ is theoretically unsatisfactory and may, in special cases, endanger protocols which happen so employ both H and H'.

This appears to have been observed by the NIST before, as the existence of the hash functions SHA-224 and SHA-384 indicates. SHA-224 is exactly identical to SHA-256, except for two things:

1. SHA-224 truncates away the 32 bits from the final output, thus turning a 256-bit hash function into a 224-bit hash function.

2. At the beginning of the hash process, SHA-224 uses a different initialization vector.

The second property means that, given a message M, the hash values SHA-224(M) and SSH-256(M) appear as two independent random values of size 224 resp. 256 bit. The relationship between the 384-bit hash function SHA-384 and the 512-bit hash function SHA-512 is similar to that between SHA-224 and SHA-256.

The SHA-2 family only consists of four different hash functions supporting four different output sizes, and it would clearly be way too tedious to define a formally independent standard for any output size s users might require.

But when defining a new standard from scratch, as is now the case for SHA-3, it is trivial to provide for the any output $s \leq n$. A generic approach is the following. Let $H(s)$ be the hash of a binary representation of $s \leq n$ and define

$$H_s(M) = \text{truncate}_s(H(s)||M).$$

With proper padding (inserting as k zero-bits between $H(s)$ and M, where k depends on the internal block size of H), this is at no cost for any user who uses H_s always with the same s. Changing s is tantamount to using a different initial value, as in the case of the SHA-2 family.

We argue that this is an easy and efficient and also secure way to support different output sizes, with security properties such as collision and preimage resistance and indifferentiability from a random oracle being provably inherited by H_s from H. The proofs are straightforward. There is one important restriction, though:

> **To avoid trivial near-collisions between $M = (H(s)||M')$ and M', one must not use H and H_s side by side. I.e., even if exactly n output bits are required, one should use H_n and not H itself.**

Thus, as easy as it is to define a flexible H_s for $s < n$ is, it has to be done as early as fixing SHA-3 itself.

Of course, if SHA-3 has been fixed as a single n-bit hash function H, and one would later need a standardized hash function with s output bits for any $s \leq n$, one could still play the same "trick" as with SHA-224 and SHA-384 by reusing entire hash function, except for a new initial value. This would give an n-bit hash function H, and a flexible hash function H'_s. H and H'_s could could safely be used side-by side ... with some obvious interoperability problems for the $n = s$ case, however.

Case 2: the native hash output is too short

Again, consider any secure n-bit hash function H. A generic way to support $s > n$ output bits via functions H_s with the property that for $i = j$ the two hash functions H_i and H_j act like independent hash functions, is to define some kind of a *counter mode.* Choose k such that one can safely fix $n * 2^k \geq s$. For any natural number i, let $\langle i \rangle$ be a k-bit binary representation of i. Define

$$H_s(M) = \text{truncate}_s((H(H(s)||M)||\langle 0 \rangle)|| \dots ||(H(H(s)||M)||\langle \lfloor s/n \rfloor \rangle).$$

This construction has still to be analyzed, but strongly conjecture that if one models H as a random oracle, one can prove H_s to be indifferentiable as well.

As the above generic construction shows, the case $s > n$ is not much different from the case $s < n$: **It is easy to provide for variable-output-size hash functions in time, but more difficult to add that property once a fixed-output-size hash function has been standardized.**

Note that some applications actually need largish hash values of size $s > n$ without demanding any improved security beyond what the underlying n-bit hash function gave. E.g., collision resistance for up to $2^{n/2}$) hash function calls could suffice. As an example, consider the Full Domain Hash (FDH) signature scheme for RSA-2048. It needs a 2048-bit hash function which ideally would have a collision resistance up to 2^{1024} – but that would be an absurd overkill, when using a 2048-bit RSA modulus. Similarly, the OAEP for RSA-2048 needs two hash functions with s and $2048 - s$ output bits. So far, such applications typically employ a first-generation hash function, which provides a small hash value. The user (this is the application programmer or the system designer) is left with finding any ad-hoc method to expand the short hash output to the required larger one.

It is hard to imagine that the next generation of hash functions should it leave to the user to find an ad-hoc method to expand the hash output for common applications such as OAEP, as the previous generation did.

3 Important Features

3.1 Hash-Based Signatures

A digital signature system can be built entirely out of hash functions, without recourse to the mathematics of finite fields or elliptic curves.

One-time signature systems were introduced by Lamport and Diffie, and improved by Winternitz; they rely on the preimage resistance of the hash function, and can sign only a single message.[2] While these systems can be useful in some contexts, such as signing a sequence of messages that can contain a chain of public keys, the tree-signature schemes developed by Merkle are more generally useful [16]. These build on the one-time signature schemes, using a tree of signature values, and relying on the collision resistance of the underlying hash function. These schemes can sign a potentially large but fixed number of messages, such as 2^{20}. More recently, multi-level tree schemes have been introduced that can sign arbitrarily high numbers of messages.

An appealing security property of hash-based signatures is their minimization of security conjectures. All practical signature schemes rely on collision-resistance, but only hash-based signatures make no other conjectures such as the difficulty of factoring, or the difficulty of the discrete log problem in elliptic curve groups. Because those other conjectures (and RSA, DSA, and ECDSA) would not hold up against a quantum computer, **hash-based signatures are the leading candidate for a post-quantum signature system.** There has been a resurgence of interest in hash based signatures in the last decade, partly because of this advantage.

Hash-based signatures are amenable to extremely compact implementations, and their performance is competitive with other digital signature technologies. Their compactness can be valuable for entity authentication in a constrained environment, e.g. 'smart objects', and for software or firmware authentication. To check a signature on a bootloader, a kernel or kernel module, or a device driver, it is necessary to have signature validation function implemented in the hardware, BIOS, FPGA, or firmware that loads and runs that software. On the signing side, these schemes have the advantage of being naturally resistant to side-channel attacks that leverage information about timing and power. Since they require only a small trusted computing base, and resisting side-channel attacks, hash-based signatures are well suited for applications requiring tamper resistance, such as protection against hardware trojans and counterfeit software or hardware. Another potential application of hash-based signatures is to signing very short messages, such as a single measured value in a sensor network.

A hash-based signature system uses its underlying hash function for several distinct purposes. A shortened hash ($s < n$) is used in the one-time

[2] See [6] for an overview and references.

signature component; the shortening reduces the size of the signatures without reducing security. In tree signatures, tree hashing is used, and a pseudorandom key derivation function is used to expand the private key into a much larger internal state. The key derivation step can be considered as the $s > n$ mode of the hash function, if the hash supports that operation. Hash-based signatures would benefit from a hash function that natively supported all three 'modes', especially in compact implementations. A SHA-3 standard that incorporated these modes would encourage and support hash based signatures.

3.2 Parallelized Tree Hashing

Tree hashing has originally been proposed in the context of hash-based digital signatures [16] to handle many one-time signatures, see above. Beyond that, however, the implementation of a tree hash function can support an arbitrary level of parallelism for hashing large messages – the larger the message, the more different threads can effectively participate. Each thread computes the hashes of one ore more subtrees. Eventually all these subtree hashes are hashed themselves to generate the final single-value output. The price for the parallelism is storage. Memory requirements grow with larger messages, even on singe-threaded implementations. Thus, parallelizable tree hashing and low-end systems don't match. A tree hash function is never compatible with an ordinary sequential hash function (except for trivially insecure hash functions, such as the xor of all message blocks). So a standard supposed to be applicable on a wide range of platforms, including low-end systems, must be based on sequential hashing.

In fact, we are only aware of a few parallel tree hashing applications. [18] describes the usage of tree hashing in peer-to-peer file sharing protocols, such as Gnutella and Direct Connect, and file sharing applications, such as Phex, BerarShare, and a few others.

But the current trend in computer architecture is that future performance improvements will largely depend on an improved parallelism, and it is hard to imagine that trend to change soon. A few years ago, a typical desktop machine had a single processor with a single core, and the main approach to a build faster machine was to increase the clock frequency. Today, a typical desktop machine has several cores, and faster machines are running at more or less the same clock frequency with more cores.

Following this trend, any demand to efficiently compute hash values for huge messages can only be satisfied by parallel tree hashing. We thus expect the practical relevance of parallel tree hashing to greatly increase over the next few years, and **a standard defining a parallel tree hashing mode for SHA-3 will be required by users.** This could be either the SHA-3 standard itself, or a sibling standard, which should soon follow after the choice of SHA-3.

In principle, one can use any secure hash function H as the building block for a tree hash function H^t, see, e.g., [4]. One has to take care about trivial collisions, however, i.e., messages $M = M^t$ with $H^t(M)$ being equal to $H(M^t)$, where M^t is determined by intermediate values and control information needed to compute $H^t(M)$. This is an issue if there is any chance an attacker might switch between hash functions.

4 Nice to Have

4.1 A Standardized way to personalize SHA-3

Given a hash function H, a hash value $H(M)$ is determined by the message M. Sometimes, however, some additional information beyond the message itself is relevant. Accordingly, the hash value should also depend on this additional information. For digital signature schemes, e.g., one usually first computes a hash $H(M)$ and then actually signs the hash value, instead of M itself (this is the common hash-then-sign paradigm). It would be useful to make the hash depend on the public signing key and related parameters. Among other things, this is some kind of defense against key substitution attacks [12] and domain parameter substitution [17]. If P is a description of the public key and the domain parameters, users could compute $H(P||M)$ instead of $H(M)$. This approach, however, looks conceptually false. The message and the additional data P are of different nature, and the above formula treats them like equals. Much worse, this could actually help attackers instead of hindering them – they could try to shift the boundary between P and M to generate trivial collisions.

A solution for this issue is the option to turn a single hash function H into different *personalized* hash functions H_P, without changing the output size. Above we rejected $H_P(M) = H(P||M)$. A generic approach to define a personalized hash function is

$$H_P(M) = H(H(P)||M).$$

But not that, similar to the other generic solutions we described above, one must never use H and H_P side by side. So if there is no personalization information, one must compute

$$H_{\langle\text{empty string}\rangle}(M)$$

instead of $H(M)$. Also, similarly to the above generic solution, this would come at no cost if there is the proper amount of padding between $H(P)$ and M.

4.2 Support for Encryption and Authenticated Encryption

Much research on block cipher-based hash functions has been motivated by low-end systems: If a system needs both encryption (using a block cipher) and hashing, then a block cipher mode of operation for hashing allows to save chip space or reduce the size of the executable.

One can turn this as well the other way round: A hash function is a very strong cryptographic primitive on its own, and a sufficiently flexible hash function could directly, or via some of its internal building blocks, support encryption and authenticated encryption. This is trivial for block cipher based hash functions: just employ any applicable block cipher mode of operation. But this is also doable for hash functions following a different design approach [5].

5 Mixing Features

A cryptographic standard has two main purposes. One is interoperability.

This issue appears to be solved once there is a standardized way to provide the features or modes we outlined above. This isnt't quite true, however. Interoperability requires a **right of way rule**, for cases where more than a single feature is required.

Users will be demand keyed hashing with support for different output lengths, for personalized tree hash functions, and so on. As an example, consider a personalized hash function for different output lengths. Combining our generic approaches could lead to either

$$\text{truncate}_s(H(H(s)||H(P)||M))$$

or

$$\text{truncate}_s(H(H(P)||H(s)||M)).$$

Either seems to be as good and secure as the other one, and we anticipate a security analysis would confirm that. Without an obvious advantage of one solution over the other, different applications will choose different solutions – defeating the purpose of a standard. Thus, beyond providing the features themselves, the standard needs to define how to combine two or more features.

6 Provable Security

As we argued above, a cryptographic standard has two main purposes. The other one is security. If the hash function isn't secure, it is useless.[3]

The current state of the art does not allow to prove a hash function being secure – not without making any unproven assumption. And even with making an unproven assumption, the performance of provably secure hash functions is not competitive. Thus, for the raw hash function one cannot expect SHA-3 to be provably secure.

The current state of the art for *modes* and *generic constructions* (of block ciphers or hash functions) is different. Reductionistic proofs of security are well-established in todays cryptography, and no mode should be acceptable without such a proof. For hash function modes that would mean that if the mode fails, then the hash function itself or one of its main internal building blocks can actually be broken. To put it simply: **If H is secure, than the mode using H is secure as well**, with *secure* denoting exactly specified security properties. No mode should be acceptable for standardization without such a proof.[4]

Using an ad-hoc mode implies the risk of being insecure, even if the raw hash function is secure. **Beyond interoperability, a benefit of standardized hash modes would then actually be their provable security – or rather, their proven security.**

7 Conclusion

Future users of SHA-3 will need much more than the raw hash function itself. They will need a message authentication mode and support to process the plaintext for public-key encryption schemes, to perform key derivation, to extract entropy from sources, and so on. They will urge for for hash-based signatures and parallel hashing modes. They will ask for personalizable hash functions and encryption schemes based on the hash function. They will combine these modes. They have the right to expect the level of proven security for the modes that is able with the current state of the art.

[3] This is a minor oversimplification – *secure* should always be qualified by ... *against this type of attack*. E.g., a hash function which fails at collision resistance but maintains preimage resistance, may not be entirely useless.

[4] For the generic constructions we gave, we believe we can provide such proofs, though we so far didn't actually try to work out all details.

Most of these issues can be dealt with by sibling standards, rather than integrating them into SHA-3 itself. However, the choice for SHA-3 determines how easy it will be to add the required modes in the future.

References

1. M. Bellare, R.Canetti, H. Krawczyk. Keying Hash functions for Message Authentication. Crypto 1996.
2. M. Bellare, P. Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. ACM Conference on Computer and Communications Security 1993.
3. M. Bellare, P. Rogaway. Optimal Asymmetric Encryption. EUROCRYPT 1994.
4. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. Sufficient conditions for sound tree and sequential hashing modes. IACR Cryptology ePrint Archive 2009: 210 (2009).
5. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications.
6. J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, M. Rückert. On the security of the Winternitz One-Time Signature Scheme. IACR eprint 2011/191, http://eprint.iacr.org/2011/191.pdf.
7. J.-S. Coron, Y. Dodis, C. Malinaud, P. Puniya. Merkle-Damgrd Revisited: How to Construct a Hash Function. Crypto 2005.
8. I. Damgaard. A design principle for hash functions. Crypto 89.
9. E. Fujisaki, T. Okamoto, D. Pointcheval, J. Stern. RSA–OAEP is secure under the RSA assumption. Crypto 2001.
10. Internet Cryptography Web Pages: Statistics, 2011, http://www.mindspring.com/ dmcgrew/ic/statistics.html.
11. A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. Crypto 2004.
12. Neal Koblitz, Alfred Menezes. Another look at security definitions. IACR eprint 2011/343, http://eprint.iacr.org/2011/343.pdf.
13. S. Lucks. A Failure-Friendly Design Principle for Hash Functions. Asiacrypt 2005.
14. A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography, Cambridge University Press 1997, http://www.cacr.math.uwaterloo.ca/hac/.
15. R. Merkle. One-way hash functions and DES. Crypto 89.
16. R. Merkle. A digital signature based on a conventional encryption function. Crypto 87.
17. Serge Vaudenay. Digital Signatures with Domain Parameters. ACSIP 2004, http://infoscience.epfl.ch/record/99523/files/Vau04b.pdf.
18. http://en.wikipedia.org/wiki/Tiger-Tree_Hash