The Long-Short-Key Primitive and Its Applications to Key Security

Matthew Cary[1], Matthias Jacob[2], Mariusz H. Jakubowski[3], and Ramarathnam Venkatesan[3]

[1] Google
[2] Nokia
[3] Microsoft Research

Abstract. On today's open computing platforms, attackers can often extract sensitive data from a program's stack, heap, or files. To address this problem, we designed and implemented a new primitive that helps provide better security for ciphers that use keys stored in easily accessible locations. Given a particular symmetric key, our approach generates two functions for encryption and decryption: The *short-key* function uses the original key, while the functionally equivalent *long-key* version works with an arbitrarily long key derived from the short key. On common PC architectures, such a long key normally does not fit in stack frames or cache blocks, forcing an attacker to search memory space. Even if extracted from memory, the long key is neither easily compressible nor useful in recovering the short key. Using a pseudorandom generator and additional novel software-protection techniques, we show how to implement this construction securely for AES. Potential applications include white-box ciphers, DRM schemes, software smartcards, and challenge-response authentication, as well as any scenario where a key of controllable length is useful to enforce desired security properties.

1 Introduction

On today's computing systems, security often relies on public-key cryptography and symmetric ciphers. Protection of cryptographic keys against eavesdroppers is a crucial issue. While the problem of keeping secret keys secret during key exchange is well understood in principle, hiding data in memory or on disk is difficult on open platforms. In particular, symmetric keys are often used for performance reasons, but support only "private" operations that must be secured against malicious observation and tampering. For example, viruses and bots can scan a user's hard drive or launch side-channel attacks on cryptographic functions [25, 30]. Widely available tools like debuggers and system monitors facilitate both manual and automated extraction of keys from unsecured storage.

In practice, software has used various key-protection approaches to create roadblocks against access to sensitive data at runtime. For example, white-boxing [8, 9] transforms keys into code that performs cryptographic operations without using key material explicitly. Obfuscation and tamper-resistance techniques [1, 2, 6, 10, 11, 21, 28] similarly help to prevent hackers from easily finding

K. Matsuura and E. Fujisaki (Eds.): IWSEC 2008, LNCS 5312, pp. 279–297, 2008.

and extracting keys. Unfortunately, such methods provide limited solutions that work only under restrictions and offer few security guarantees or analyses.

In this paper, we approach the problem from a different angle and implement a secure cryptographic system based on our *long-short-key primitive (LSK)*. LSK is a universal tool to address concerns with easy key extraction and distribution. Like white-boxing, our method generates code for cryptographic operations with a given key. However, instead of obfuscation, we transform the key into an arbitrarily long data stream that is used to implement encryption and decryption. This transform is one-way, so that the original short key cannot be efficiently reconstructed from the long version. Moreover, the long key cannot be easily compressed, forcing a lower bound on the size of any hack derived from the long key. The idea of LSK is related to theoretical approaches to bounded storage and retrieval [5, 14]; however, earlier work has not applied such techniques to the problem of keys exposed during cryptographic operations.

More specifically, our approach accepts a symmetric key as input and generates two corresponding functions for cryptographic operations using that key:

- The *short-key* function uses the original unprotected cipher key for efficient encryption and decryption. The secret key may be either incorporated into the function code or passed in as a parameter, so that a single short-key function can suffice for all keys. Typically compact and efficient, this function's implementation should run only in a secure environment, such as a protected server or tamper-resistant smartcard.
- The *long-key* function is operationally equivalent to the short-key version, but uses an arbitrarily large data block derived from the short key. The size of the long key is a security parameter. The owner of the short key generates the long key from the short key and an optional secret. Given the long key, efficient recovery of the short key is not possible. The entire long key is required for encryption and decryption of data in general, though a particular input text may use only a portion of the long key.

The main goal of our approach is to provide a guaranteed security property, namely minimum effort needed to analyze a long-key function, as well as to extract, store and reuse a long key. This has various applications, such as protection of long-lived symmetric keys in DRM systems and secure smartcard simulation in program code where key exchange is not possible. Usage scenarios extend beyond encryption; for example, a challenge-response authentication protocol can depend on a server's short key to verify a client's possession of a long key (e.g., via periodic requests for random blocks of that key).

Our method does not fully address all possible security issues with key protection. For example, an adversary can still copy a long key, as well as simply call a long-key function to perform encryption and decryption. However, such situations are beyond the scope of this paper. As in bounded-data models [5, 14], we assume that an adversary has restricted storage capabilities, or at least limited bandwidth for data retrieval. We analyze security mainly in this context.

Depending on usage scenario, obfuscation of the long key may be useful to complicate the task of extracting key bits and calling the long-key function. For

example, one may build a platform-specific long-key implementation bound to a particular application via additional mechanisms. This paper also presents some novel obfuscation techniques to increase the attacker's workload, but our basic method does not depend on obfuscation for its main security property.

Our approach focuses on symmetric block ciphers such as DES and AES, but is applicable to stream ciphers and public-key systems as well. More generally, the duality of long and short keys is of independent interest in various cryptographic scenarios, including protocols and authentication.

The following is a summary of our contributions in this paper:

- We define a new cryptographic construction, the *long-short-key primitive*, and propose its usage as an alternative to other key-protection methods in some scenarios.
- Using a cryptographic pseudorandom generator and a standard block cipher, we show that an LSK scheme can be implemented in practice with provable security (equivalent to breaking the generator and cipher).
- We present several applications, including white-boxing, DRM, software smartcards, and challenge-response authentication.
- We explain some LSK-specific obfuscation techniques and describe an AES-based implementation of the LSK primitive with reasonable performance penalty.

1.1 The Case for Long Keys

The long-short-key primitive can significantly improve security in various scenarios, as described later. Below we summarize several applications:

Block-Cipher Security. Since the short key may change on every block-cipher operation, our construction significantly improves security of encryption. For example, it becomes difficult to carry out the types of ciphertext-correlation attacks possible in ECB mode. In addition, the long key can improve security of white-box ciphers, which aim to protect keys by hard-coding them into programs.

Digital Rights Management. On modern computing systems, reverse engineers and hackers can typically inspect contents of memory and disk. Widely accessible tools, such as debuggers and system monitors, allow sophisticated users to access runtime state and files, including sensitive data like cryptographic keys and authentication credentials. Even when hardware enforces data protection, security exploits and side-channel attacks [29] may still render secret bits open to observation. Such transparency creates problems for various popular applications that need to protect cryptographic keys. For example, DRM systems decrypt content on PCs, enabling attackers to extract symmetric keys from memory. Since such keys are quite short (e.g., 128 or 256 bits), they can be easily shared across different devices and distributed via pirate channels. In general, symmetric-key encryption usually relies on well known, standardized ciphers that allow easy reuse of keys lifted from memory. A leaked key may even be typed in by a PC or device user. Using LSK, however, a long key can be megabytes or possibly gigabytes in size, preventing such attacks.

Software Smartcards. In many cases, it is desirable to implement software smartcards that perform the same functionality as their hardware counterparts. While hardware can make key extraction very difficult via physical tamper-resistance, this is not possible in open software. A smartcard key extracted from software enables hardware forgery of the smartcard. Using LSK, however, it is possible to have a software long key from which it is difficult to extract the short smartcard key. The adversary is forced to extract the code that carries out the encryption or other key-based operations. Moreover, the long key may not be practical to include or use on a smartcard, preventing easy hardware forgery.

Remote Timing Attacks. Side-channel attacks [29], such as remote timing attacks, are a common threat on the Internet. By simply measuring the time required by a remote node to encrypt selected messages, it is possible to derive the full secret key. If the remote node encrypts under LSK, enough randomness is added to encryption timing, complicating implementation of remote timing attacks.

Challenge-Response Authentication. In a standard challenge-response protocol, the challenger picks a set of arbitrary variables presented to the responder. The responder is authenticated only upon presenting the correct answer to the challenger. Many challenge-response protocols use cryptographic techniques, with the responder needing to encrypt a nonce under a specific key. When the key leaks, an adversary can always answer the challenge. With LSK, however, key leakage is less likely, depending on key length and any additional key-hiding measures.

2 Security Model

For most considerations in this paper, we assume an open platform such as a PC. The user has full access, and can use tools like debuggers and disassemblers to reverse engineer code and inspect memory. When we discuss remote attacks, a weakened model applies: The adversary is able only to trigger encryptions remotely and measure the time it takes to return results.

As a building block, we use a presumably secure cipher (e.g., AES). The adversary has access to the public code of the cipher, but not to the secret key. However, when attacking the cipher, the adversary is able to carry out side-channel attacks such as fault injection and timing analysis.

In addition, we assume that storage size is bounded, and an adversary can copy only a limited amount of data at a time (similar to [5, 14]). Storage size and bandwidth may vary, depending on the attack scenario – e.g., a virus transmitting data remotely versus a local attacker using a debugger to retrieve data.

3 The Long-Short-Key Primitive

The LSK primitive stipulates a short key k and a long key l. Both keys can be used to encrypt plaintext x such that ciphertext $c = E'_k(x) = E'_l(x)$. The

long key l can be derived from the short key k. However, given the long key l, efficient retrieval of the short key k is not possible. In reality, the long key can be hundreds of megabytes, whereas the short key may contain only 128 or 256 bits, as in AES.

This paper focuses on LSK-based encryption schemes built with a block cipher. The encryption and decryption operations can be implemented in various ways, each of which has its own advantages and disadvantages. In every scheme, the encryption function requires an initialization vector IV:

- *Sequential key-block encryption* uses a sequence of long-key blocks to encrypt plaintext x. The sequence is predefined, and depends only on the IV and the length of the plaintext.
- *Counter-based key-block encryption* uses a random sequence of long-key blocks as encryption keys for consecutive plaintext blocks, but without dependencies on ciphertext.
- *Selective key-block encryption* selects a random block of the long key and passes this to the cipher, retrieving the next random key block based on the resulting ciphertext.

3.1 Long-Key Construction

One secure implementation mechanism for LSK involves using a cryptographic generator to derive the long key, with the short key serving as a seed; we then treat blocks of the long key as separate symmetric keys. More specifically, we construct the N-bit long key l from a short key k by using a cryptographic pseudorandom number generator or stream cipher $R(k, N)$. R accepts k as a seed and generates N pseudorandom bits. For efficiency, R can be a random-access stream cipher, such as a block cipher in counter mode, which enables generation of an arbitrary long-key block without first computing any other long-key parts [20]. Encryption and decryption use blocks of the long key-stream $l = R(k, N)$ as individual cipher keys. Since these operations use the original standard cipher $E()$ as a building block, cipher security is preserved.

After Alice computes the long key l from the short key k, she sets up encryption by handing a portion of l to Bob. During the encryption, l serves as the 'code book' for Alice and Bob. When Alice encrypts a plaintext message m, she computes $c = E_{k'}(m)$, where k' is the j-th sub-key of the long key (for some index j). She then sends (c, j) to Bob. Bob looks up the j-th sub-key of the long key l and decrypts c. Alice is much more powerful than Bob in this setting, since she can derive any long-key portion from her short key k, whereas Bob is tied to the long-key fragment he received. He is able to encrypt as well, but only within his set of sub-key blocks. When Alice decides to black-out Bob, she simply switches to a different range of sub-keys in l. The following paragraphs will show how important this seemingly simple technique can be in implementations.

Operation Modes For LSK. In our construction so far, Alice is solely responsible for security, since she can pick which parts of the long key l to use

for encryption. In the worst case, she could degrade the whole long-key security to the cipher's security – i.e., when she picks only the first long-key block every time she encrypts a plaintext message. To eliminate this risk, we present three different mechanisms that enhance the security of the long-key encryption scheme.

Sequential key-block encryption. One scheme is to use consecutive long-key blocks as keys. In the first ciphertext message, Alice sends to Bob the pair (c, i), where c is the ciphertext and i indicates the i-th long-key block used as the encryption key; in all consecutive messages, she sends only c. Bob automatically uses blocks $i+1, i+2, ...$ for decryption. Alternately, Alice can simply encrypt each plaintext with long-key blocks $0, 1, 2, ...$

Counter-based key-block encryption. In the counter-based scheme, Alice encrypts sequences of numbers $i, i+1, i+2, ...$, using the cipher E_{IV} under some initialization vector IV. When she sets up the communication, she sends IV to Bob. When she encrypts the first data block, she uses key-block with index $E_k(i) \mod N$ from the long key; for the next data block, she uses $E_k(i+1) \mod N$, and so on. Bob also computes the same sequence of block indices and decrypts the messages accordingly. For a simpler scheme, i can begin at 0, and the first long-key block can be used as the initial value IV.

As compared to sequential key-block encryption, the advantage of the counter-based method is less locality during encryption. To decrypt a ciphertext, an adversary needs to obtain arbitrary parts of the long key l. In addition, this method has a random-access property for decryption; i.e., any block can be decrypted independently of other blocks.

Selective key-block encryption. Similar to CBC mode in block ciphers, the next key block to use from the long key can also be decided based on the most recent ciphertext and key. Alice starts encryption using some block k_j of the long key (e.g., the first block or a block determined by an initial value). She then computes the first ciphertext block c_0 based on key block k_j and first plaintext block p_0. The next key-block index is then $c_0 \mod N$. In general, for $i > 0$, the $i-$th key-block index is $c_{i-1} \mod N$, which encrypts the $i-$th plaintext block p_i.

Selective key-block encryption has a significant advantage over the counter-based method, since the former uses a different key block combination for different plaintexts. Therefore, this mode randomizes the key block accesses best and requires an adversary to obtain all long-key bits in general.

Depending on the random distribution used in an implementation, key blocks in both counter-based and selective key-block encryption are potentially scattered across the entire long key. In addition, different random subsets of the long key may be used for operations with each specific short key. With the selective key-block encryption scheme, the long-key subsets also depend on the input text. This provides better protection against an adversary who has partial knowledge of the long key. Also, this defends against side-channel attacks, such as cache-timing analysis. However, these methods could be slow because the random memory accesses may cause many cache misses.

Sequential key-block encryption simply uses consecutive long-key blocks to process consecutive text blocks, wrapping around whenever text size exceeds long-key size. This is a simple strategy that does not offer the capability to use random subsets of the long key. The apparent trade-off that we need to analyze more carefully is the security through randomization vs. the performance penalty due to random memory accesses.

4 Applications

4.1 Block-Cipher Security

A typical problem with block ciphers is susceptibility to correlation attacks in ECB mode. If used in CBC mode, ciphers must be synchronized. This often causes problems in unreliable communication channels, such as UDP-based transmissions. LSK improves upon ECB without this drawback of CBC, because an adversary without knowledge of the long key cannot correlate identical ciphertext blocks, even if Alice uses a simple operational mode and sends the long-key-block index on every transmission.

In addition, LSK improves the security of white-box block-cipher implementations. Often it is possible to carry out fault-injection attacks (e.g., [22]) to extract the key from the white-box cipher. With support from the long key, this attack becomes difficult, because the cipher key may change on every block encryption.

4.2 Digital Rights Management

A main goal in DRM is to protect a secret key k of a common publicly known cipher $E_k(x)$ on an open platform (such as Microsoft Windows). After k leaks, an attacker can decrypt any ciphertext c using $x = D_k(c)$. To determine k, an attacker can use disassemblers, debuggers and other reverse-engineering tools that help investigate memory during program execution. When k is located within a single stack frame or memory location, identifying and extracting k may pose few difficulties. Similarly, side-channel attacks like cache-timing analysis and fault injection can isolate k, which may comprise only a few bytes.

With a long key, an attacker's tasks become more difficult. First, the key k does not fit into a single stack frame or memory location. Secondly, side-channel attacks are significantly more difficult when an attacker has more key bits to retrieve. Also, common white-box techniques are more effective when applied to a cipher that uses a large amount of key material.

To ensure security, it is also important that the construction of the long key be transparent and derivable from common cryptographic primitives. Our present work does not generate a new cipher, but uses existing symmetric encryption to create a new cipher construction that has better security properties for implementation on open platforms.

4.3 Software Smartcards

Smartcards are a ubiquitous technique for authenticating users. It would be desirable to have a software pendant for hardware-based smartcards on PC platforms – not only to avoid proliferation of hardware readers, but also to streamline smartcard updates. One problem, however, is to protect smartcard-specific secrets in software. When the user types in a PIN code, smartcard software typically uses this to compute a ciphertext validated against a stored ciphertext. When smartcard-specific secrets leak, an adversary can forge PIN codes and compromise the system.

On an open platform, LSK hides the short smartcard secret inside the long key. Smartcard-simulator software runs in selective key-block mode and contains only the long key. When the user types a PIN, the software encrypts the PIN via the LSK construction, and compares this to a stored ciphertext. There is no need to secure a separate secret on the open platform, while hardware tamper-resistance protects the short key in the smartcard.

4.4 Remote Timing Attacks

In remote timing attacks (e.g., [4]), an attacker exploits timing properties of an encryption algorithm (e.g., RSA-CRT) to extract secret keys. Typically, these attacks can be countered by blinding the input data. This additional randomness, which is unknown to the adversary, forces the encryption algorithm to process a different plaintext message. In many encryption schemes, such as RSA, blinding can be inverted after encryption by exploiting algorithmic properties. For the adversary, the key becomes hard to extract, while the performance penalty is small. However, blinding does not work on all encryption schemes, because it may be hard to revert the blinding operation without handing out the secret randomness to every receiver. For remote attacks to be successful, the adversary must measure the timing of a certain number of encryptions under the same key. An LSK-based scheme fixes this problem, because LSK uses a different cipher key for every encryption. Given that the long key can comprise a few hundred megabytes, this will stretch the duration of the attack significantly, if not render the attack practically impossible.

4.5 Challenge-Response Authentication

In cryptographic challenge-response authentication protocols, the key must not leak. If it does, an adversary can easily answer any arbitrary cryptographic challenge from the verifying party. Using an approach based on LSK, the challenger can simply hand the long key to the responder and later request parts of the long key, combining this with encryption of nonces to provide better security. For an adversary, it is virtually impossible to answer challenges without knowing the entire long key, which is difficult to extract because of its size and any additional protection measures.

5 Implementation

An important part of this work is implementation of LSK-based schemes in a secure manner. In some contexts, a problem stems from an adversary's ability to copy the long key onto media like DVDs or over the network, transferring the key to another machine. Therefore, the long key should also be hidden in software code. Extracting portions of the key may be feasible, but obtaining the entire key should be difficult.

In practice, choosing an appropriate length for the long key will depend on the application. If the goal is mainly to prevent humans from memorizing or writing down keys, even several hundred bytes or a few kilobytes of long key may suffice. If we wish to make electronic key distribution unwieldy, several hundred megabytes or more may be the minimum if the Internet or CD/DVD media are involved. Advances in networking and storage capabilities may affect the required long-key size as well. In general, the protection designer should carefully consider aspects such as hardware and software capabilities, as well as security requirements in particular scenarios.

We have implemented a construction we call *ExAES*, which is an LSK-based scheme with AES as a cipher. In ExAES, we expand the secret AES key into a long key using a pseudorandom number generator. We then replicate the AES code, embedding blocks of the long key in each replica. Finally, we merge the different AES cipher implementations to hinder attacks that attempt to extract portions of the key.

5.1 Iterated Obfuscation

As a novel technique, we use *iterated obfuscation* to establish a general framework for the implementation of our obfuscation techniques. This methodology involves the iterated application and recombination of various obfuscating transformations (or *primitives*) over code, with the output of each successive transformation serving as input to the next. Via this strategy, even simple and easy to implement primitives can be cascaded to yield effective obfuscation.

As an example, the technique of oblivious hashing (OH) [7] can serve as an obfuscation primitive. A single OH transformation injects code to hash a program's runtime state (i.e., data and control flow), thus ascertaining execution integrity of the original code. Applying OH again to the transformed program protects both the original program and the first OH round. In general, each new OH round verifies the integrity of both the original program and all previous OH rounds.

To increase security further, arbitrary other primitives can be combined and iterated with the OH rounds. For ease of design and implementation, such primitives can be quite simple – e.g., conversion of variable references to pointer references, and even source-to-source translation among different code dialects or languages. Via iteration, the interaction of simple primitives can lead to emergent code behavior and achieve the effect of far more complex obfuscation operators.

This is related to both iterative complex systems (e.g., cellular automata) and iterated rounds in cryptographic constructions.

5.2 Generic Block-Cipher Obfuscation

In this section, we define an obfuscation operation $O'(\cdot)$ intended to compose two different cipher keys. This operation transforms a concatenated block cipher $E_{k_0} \cdot E_{k_1}$ into an obfuscated sequence $O'(E_{k_0} \cdot E_{k_1})$ in which it is difficult to isolate E_{k_0} and E_{k_1}. Both ciphers are obfuscated individually on a per-round basis, but have the same security as the original. We assume that the per-cipher obfuscation is weak, so that key bits can be extracted in constant time from every round.

Interleaving Rounds. To combine two cipher instances, we interleave their rounds. As in shuffling two decks of cards, we pick cipher rounds randomly from each cipher, with the long key serving as a source of randomness. Formally, given $E_{k_0} = R^1_{k_0} \cdot R^2_{k_0} \cdot \ldots \cdot R^n_{k_0} = \cup_{i=1}^{n} R^i_{k_0}$ and $E_{k_1} = R^1_{k_1} \cdot R^2_{k_1} \cdot \ldots \cdot R^n_{k_1} = \cup_{i=1}^{n} R^i_{k_1}$, we construct $E_{k_1} \odot E_{k_2} = \cup_{i=1}^{2n} R^{\rho(i)}_{k_{\pi(i)}}$, where $\rho(i)$ is a permutation and $\pi(i) \in \{0, 1\}$ randomly selects k_0 or k_1.

Weakly obfuscated, the rounds themselves remain in the same order as in the original ciphers (i.e., $\rho(i) < \rho(i + k)$ when $\pi(i) = \pi(i + k)$). The computed ciphertexts may be different than in the original cipher, depending on the interleaving. However, security remains unchanged, since the same round functions are used. In the appendix, we describe methods to merge and obfuscate the rounds themselves.

5.3 Performance

A main trade-off in our white-box is the number of rounds for the long key. This number is significantly larger than the number of rounds in a normal cipher. To gauge the effect, we measured the number of cycles consumed for every encrypted byte with different numbers of rounds and different obfuscation levels.

Figure 1 shows the results. The additional performance overhead becomes superlinear relatively early when the obfuscation level becomes larger. However, these additional operations are necessary to disguise round boundaries.

We also compare our white-box (WB) to AES and RSA (from Microsoft's Crypto API) in terms of performance. We chose a key size of 512 bits in all cases.

Figure 2 shows the performance comparison with AES and RSA in cycles per encrypted byte. The compact white-box implementation with the short key and without any additional obfuscation is only slightly slower than AES. However, even with obfuscation turned on, our white-box is still faster than RSA. (Involving symmetric and asymmetric encryption, this comparison is only for illustration.)

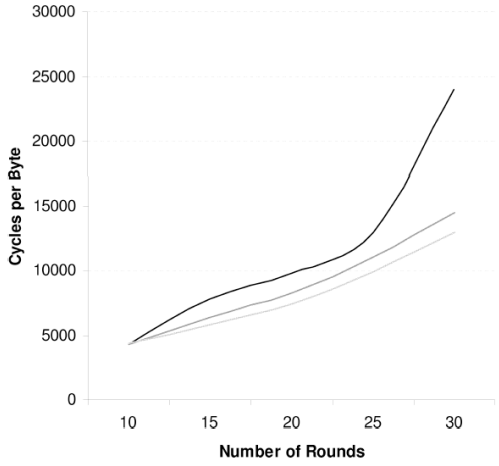

Fig. 1. White-box performance when increasing the total number of rounds and the obfuscation level

Algorithm	Cycles per Byte
AES	200
RSA	2000
WB Compact	202
WB	1500

Fig. 2. Performance comparison between WB, AES, and RSA

5.4 Security Assessment

Our obfuscation algorithm does not depend on precise types and quantities of primitives. Due to space constraints, we have omitted such specifics in our description. However, when analyzing the security of our algorithm, we assume the adversary has complete knowledge of all obfuscation primitives used; we do not assume any security-by-obscurity in choice of the primitives. The adversary does not know the random sequence which determines the application of the primitives, as such knowledge would be tantamount to knowing the private key.

The powerful side-channel and fault-injection attacks [22] mentioned in §6 show that round boundaries in an obfuscated cipher must be difficult to detect. Below we provide an example of an attack on ExAES, given knowledge of round boundaries. The problem is that each round is composed of several distinct types of operations. For example, the AES and ExAES ciphers used here alternate variable XOR operations with permutation and field operations implemented via table lookups. The operations must be applied sequentially; thus, if the distribution of hardware instructions reveals this alternation, the round boundaries could be discovered and the cipher broken.

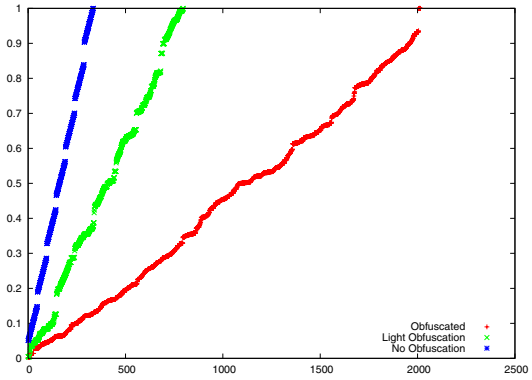

Fig. 3. Effect of obfuscation on round boundaries. Note the visible segmentation in the leftmost curve, which clearly indicates round boundaries. As more obfuscation is applied, the segmentation becomes less apparent, thus hiding the boundaries.

As a concrete example, consider the Mix operation in AES. The multiplication in the AES field is implemented by a lookup table indexed by data bytes whose entries are XORed together to produce the result. To implement this in hardware, XOR instructions with both operands variable are used, in contrast to XOR instructions with static constants that are used to mix in key data before applying the S-Box. These variable XOR instructions may be discovered statically and their cumulative total plotted against the number of instructions to produce a cumulative distribution curve.

This has been done in Figure 3 with various levels of obfuscation applied. The figure plots the cumulative distribution of variable XOR instructions by total number of instructions with various levels of obfuscation applied. Plots with higher levels of obfuscation are flatter, as the total number of instructions is higher. The key point is to see how the round structure is clearly visible in the non-obfuscated version (leftmost curve), and that structure disappears as the obfuscation level is increased (middle and rightmost curves).

Attack Against a Single Round. To illustrate the importance of hiding round boundaries, we show how a single ExAES round may be attacked even if no information is known about the byte permutation or the S-Boxes. We note that an attack against a round of AES is even simpler, since the key is involved only in the XOR performed before the S-Box.

The crux is that the Mix operation mixes only four bytes at a time. Hence, by comparing the action of the round on inputs that vary at single bytes, the permutation can be discovered up to the order within each column. After the Mix operation, unknown dummy operations will XOR with each column to produce the output, 32 bits in total. The Mix operation itself is known. By guessing the 32 bits of dummy XOR after the Mix operation, then guessing which of the 24 possible permutations were used on the input identified as forming the column, we arrive at a guess of the state of those bytes immediately after the

S-Box operation. Varying each input byte to reconstruct the S-Box and confirm the guess completes the attack against that column. Repeating this for the remaining columns reveals the rest of the round key.

This attack takes approximately 2^{40} work, which is very feasible on a modern processor. The attack can recover the several thousand bytes of key that are used in the case of independent random S-Boxes, in addition to the permutation and final layer of dummy operations. Further improvements to this attack only underscore the need for an obfuscator to hide fully the round boundaries of the cipher.

6 Related Work

Software obfuscation is a well known problem that has never seen a comprehensive solution. [10] is a systematic taxonomy of obfuscation techniques in the context of Java bytecode. Many of these apply to our problem, and are incorporated into our system. In particular, we use data obfuscation by aggregation and ordering, as well as control-flow obfuscation; see [10] for definitions. Some of the obfuscation techniques in [10] are used with complex control flow, data structures or procedure networks, and do not apply to the less general problem of white-boxing ciphers that we consider here. The extension of [11] describes *opaque predicates*, which are used for obfuscated control flow. We use a limited form of these in our current version and will incorporate more complex constructions in future versions.

Other approaches to secure computation include secure circuit evaluation [16, 18, 23, 32, 33]. In this model, several parties hold separate inputs to a common function, and want to compute the function without revealing any information about their inputs other than what is contained in the function output. The multiparty-communication nature of this problem produces solutions inappropriate for our white-box model. In particular, protocols for secure circuit evaluation rely on random choices that must change with each run of the protocol. In a white-box context, an adversary is able to reset any state, including random counters within the attacked program.

The specific obfuscation problem of creating a white-box cipher (DES) was addressed in [8, 9]. This was later attacked using fault-injection techniques [22, 27]. The idea of [8, 9] first hard-codes the DES round keys into the round computation. The DES encoding is then seen as a series of functions – some affine, while others entirely non-linear and implemented with lookup tables (the S-Boxes). Random invertible functions are chosen, paired with their inverses, and inserted into the series of DES-related functions. The encoding operation is then reassociated to combine the random functions with the DES steps, expanding the affine functions into lookup tables as appropriate. The goal is to hide the key operations with the random invertible functions.

The construction in [9] was attacked in [22] by injecting faults before the last round to probe for key bits. The essential point is that once the boundaries can be discovered, the round function can be taken apart in spite of the obfuscating

random functions applied to it. Their construction differs from ours in being a very specific algorithm—table composition and re-association—applied at a high-level to the cipher. As described below, our method is applied across all levels of white-box program generation, from the high-level representation of the operation sequence to the the low-level access of individual bytes.

Much work has been done in areas of distributed execution and security of mobile code [17, 24]. This often focuses on running untrusted programs on trusted machines, which is opposite to our problem of running a trusted program on an untrusted machine. A related area is software tamper-resistance [26, 31], where the problem is to guarantee that distributed code is executed unmodified on untrusted machines. In white-box applications, it is immaterial whether or not the white-box code is modified, provided that the output is correct. The only aim is to compute while keeping a secret hidden. Many tamper-resistance solutions implicitly rely on keeping obfuscated secrets, such as locations of checksumming that occurs in code. Software white-boxing addresses this secret-hiding explicitly and does not attempt wider security goals.

Recently, side-channel attacks have been shown to be effective against even software implementations of ciphers [3, 4, 29]. In cache-timing attacks, an adversary detects the number of cache misses during encryption operations and reconstructs the key bits based on this information. In particular, ciphers like AES that access memory locations based on the values of the key are prone to this attack. Our white-box protects against these attacks because of the long key and the additional amount of randomness in the cipher algorithm.

The extended limitations on obfuscation presented in [19] suggest that obfuscation is more difficult in the presence of auxiliary dependent information. An example of this in the context of our system would be an adversary who has access to two white-box versions of the cipher with a particular key. While we believe our system should be practically secure against this attack, a detailed analysis remains to be performed.

Finally, there is a large body of work on cryptographic algorithms for bounded storage [5, 12, 13, 14, 15]. Typically, the security framework is similar to our LSK platform, with the exception of remote attacks in bounded-data models (e.g., a virus sending secret data over a network with limited bandwidth).

7 Conclusion

We proposed a new construction, the long-short-key primitive, that hides keys and improves implementation security of block ciphers. Our method creates a compact, efficient implementation of encryption, along with white-boxed long-key decryption code that may be arbitrarily large, as controlled by the user. Unless a cryptographic pseudorandom generator is broken, any hack of the white-box must have a provably large size to recover the full long key. In essence, we provide a private encryption function that uses a short key, along with a corresponding public decryption function that requires a long key. In addition, different encrypted content requires different sections of the long key for decryption,

so that breaking the obfuscation enough to decrypt one ciphertext does not necessarily allow decryption of others. The key sections required for decryption may be revealed only as decryption proceeds.

Our method has a number of practical applications, including DRM key management and software-based smartcard simulators designed to hide a short key present on tamper-resistant hardware smartcards. Our main provable security metric, namely the minimum size of any white-box hack, is of independent theoretical interest as well. While our system provides a symmetric cipher, the novel asymmetric paradigm of short "private" keys and arbitrarily long "public" keys may find other applications that currently rely on true asymmetric cryptography.

References

1. Aucsmith, D.: Tamper resistant software: An implementation. In: Anderson, R. (ed.) IH 1996. LNCS, vol. 1174, pp. 317–333. Springer, Heidelberg (1996)
2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
3. Bernstein, D.J.: Cache-timing attacks on AES, `http://cr.yp.to/papers.html#cachetiming`
4. Boneh, D., Brumley, D.: Remote timing attacks are practical. In: USENIX Security Symposium (2003)
5. Cash, D., Ding, Y.Z., Dodis, Y., Lee, W., Lipton, R., Walfish, S.: Intrusion-resilient key exchange in the bounded retrieval model. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 479–498. Springer, Heidelberg (2007)
6. Chang, H., Atallah, M.J.: Protecting software code by guards. In: Digital Rights Management Workshop, pp. 160–175 (2001)
7. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious hashing: Silent verification of code execution. In: Proceedings of the 2002 Information Hiding Workshop (October 2002)
8. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.: White-box cryptography and an AES implementation. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595. Springer, Heidelberg (2003)
9. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.: A white-box DES implementation for DRM applications. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 1–15. Springer, Heidelberg (2003)
10. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand (July 1997)
11. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages, POPL 1998, pp. 184–196 (1998)
12. Di Crescenzo, G., Lipton, R.J., Walfish, S.: Perfectly secure password protocols in the bounded retrieval model. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 225–244. Springer, Heidelberg (2006)
13. Dagon, D., Lee, W., Lipton, R.: Protecting secret data from insider attacks. In: Proceedings of Financial Cryptography (2005)

14. Dziembowski, S.: Intrusion-resilience via the bounded-storage model. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 207–224. Springer, Heidelberg (2006)
15. Dziembowski, S.: On forward-secure storage. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 251–270. Springer, Heidelberg (2006)
16. Feige, U., Killian, J., Naor, M.: A minimal model for secure computation (extended abstract). In: STOC 1994: Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing, pp. 554–563. ACM Press, New York (1994)
17. Gao, D., Reiter, M.K., Song, D.X.: On gray-box program tracking for anomaly detection. In: USENIX Security Symposium, pp. 103–118 (2004)
18. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC 1987: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing, pp. 218–229 (1987)
19. Goldwasser, S., Kalai, Y.T.: On the impossibility of obfuscation with auxiliary input. In: FOCS 2005: Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (2005)
20. Dj Golić., J.: Stream cipher encryption of random access files. Information Processing Letters 69(3), 145–148 (1999)
21. Horne, B., Matheson, L.R., Sheehan, C., Tarjan, R.E.: Dynamic self-checking techniques for improved tamper resistance. In: Digital Rights Management Workshop, pp. 141–159 (2001)
22. Jacob, M., Boneh, D., Felten, E.: Attacking an obfuscated cipher by injecting faults. In: ACM CCS-9 Workshop (DRM) (2002)
23. Kilian, J.: A general completeness theorem for two party games. In: STOC 1991: Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing, pp. 553–560 (1991)
24. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: USENIX Security Symposium, pp. 191–206 (2002)
25. Kocher, P.: Timing attacks on implementation of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109. Springer, Heidelberg (1996)
26. Lie, D., Thekkath, C.A., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J.C., Horowitz, M.: Architectural support for copy and tamper resistant software. In: ASPLOS, pp. 168–177 (2000)
27. Link, H., Neumann, W.: Clarifying obfuscation: Improving the security of white-box encoding. Cryptology ePrint Archive Report 2004/025 (2004)
28. Lynn, B., Prabhakaran, M., Sahai, A.: Positive results and techniques for obfuscation. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 20–39. Springer, Heidelberg (2004)
29. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
30. Shamir, A., van Someren, N.: Playing hide and seek with stored keys. In: Financial Cryptography (1998)
31. van Oorschot, P.C., Somayaji, A., Wurster, G.: Hardware-assisted circumvention of self-hashing software tamper resistance. IEEE Transactions on Dependable and Secure Computing 2(2), 82–92 (2005)
32. Yao, A.C.: Protocols for secure computations. In: FOCS 1982: Proceedings of the Twenty-third IEEE Symposium on Foundations of Computer Science, pp. 160–164 (1982)

33. Yao, A.C.: How to generate and exchange secrets. In: FOCS 1986: Proceedings of the Twenty-seventh IEEE Symposium on Foundations of Computer Science, pp. 162–167 (1986)

A Merging Rounds

When block-cipher rounds are simply shuffled, a brute-force method could successfully rearrange rounds in the original ciphers. In addition, various other key-extraction attacks are possible if round boundaries are easily discernible. In this section, we present a list of operations that preserve semantics but enhance security by merging operations of different rounds.

Our approach obfuscates an operation sequence by applying a series of obfuscation transformations. Most of these are peephole; i.e., they are applied locally to small groups of instructions, usually pairs. The transformations are organized below into several classes. When variously composed and iterated, the combined transformations will increase the complexity of the obfuscation more than the sum of each individual transformation. As attacks against our obfuscation develop, new transformations will be become apparent and can easily be integrated into our system.

We combine these transformations randomly to obfuscate code. Some tuning is required to maximize the avalanche effect of the obfuscations while minimizing the increase of program size. For example, dummy permutations, which translate into a large number of instructions, are inserted sparsely into rounds between commuting and combining transformations, which have a smaller impact on code size. This heuristically maximizes the amount of diffusion from each permutation while minimizing the total number of permutations performed.

Space constraints do not permit us to describe all transformations in detail. We will elaborate on several illustrative techniques and leave the remainder general. An advantage of our obfuscation method is that the exact transformations used are not critical; we believe most reasonable simple ones can be used iteratively to produce effective obfuscation.

Our machine model assumes a simple processor operating over a random-access array. We view a program is a sequence of operations over a logical array of bits. The operations are only those needed to implement the cryptographic functions of interest, such as bitwise Boolean operations, permutation, copying, addition, multiplication, and some simple control flow. We do not expect the operations to be universal in the sense of ability to describe any computation. This operation set was chosen as the smallest sufficient to compute AES efficiently, as well as introduce execution that cannot be analyzed in a purely static way. In particular, while control flow is not strictly necessary to produce an AES-like program, this is useful in obfuscation when combined with opaque predicates [11].

Simple Transformations. Our first class of obfuscations is simple transformations that do not cause internal changes to any operation. For example, a random permutation of a subset of the working data can be applied, followed by

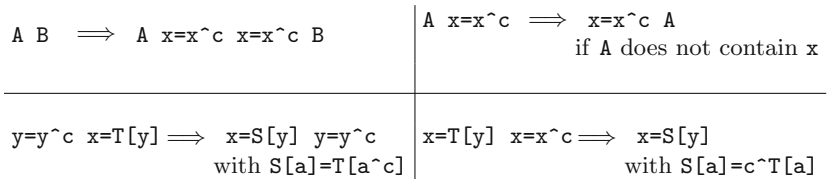

Fig. 4. Some simple obfuscation transformations. In the above, ^ denotes XOR, lower-case letters are working data locations, upper-case letters signify generic operations, and braces are used to denote table lookup.

a permutation that moves the working data back to its original location. Similarly, two XOR operations with the same random constant can be performed on a subset of the working data. Both these operations may seem nonsensical, but become quite powerful when combined with other operations below. We also note that if the input and output sets of an adjacent pair of operations do not intersect, they may be commuted. That is, the sequence of two operations `[ A B ]` may be replaced by the sequence `[ B A ]`.

Morphing Transformations. Techniques in this class of obfuscations commute two operations in ways that cause the operations, but not their number, to change. For example, an operation and a permutation may be commuted by modifying the operation to apply the permutation to its inputs and outputs. The permutation or its inverse is applied, depending on whether the operation precedes the permutation or vice-versa. This makes the simple transformation above useful. Several more complicated ones are shown in Figure 4.

Lookup-Table Obfuscations. These transformations apply to lookup tables. The above transformations can take a single lookup table from an unobfuscated version of a program and create several obfuscated versions, each permuted and changed. While this increases the obfuscation, it also leads to a large amount of static data. Furthermore, because most of the tables are byte-oriented, they are vulnerable to exhaustive analysis of all 2^{16} possible modifications.

We use two techniques to address these problems – *gradual lookup-table correction* and *dummy lookup tables*. In the former, the lookup tables are stored with errors that are corrected and changed as the program executes. The latter allows for dummy operation sequences, described below, to use actual lookup tables for computation interleaved with the error correction. This obscures which values in the lookup tables are correct and which are erroneous.

B Low-Level Obfuscation Techniques

Our low-level obfuscations hide patterns of data access by operations independently from the semantics of the operations. After these obfuscations, the code generated for a byte-array read or write is more complicated than a simple lookup into the array.

An example of this technique is byte-array indirection. Logical bytes are stored discontinuously throughout the lookup tables, which are changed over the course of the program's execution. A set of indirection indices is used to look up the location of each byte. These indices are spread discontinuously throughout the lookup-table data. As their locations are known statically at generation time, moving indices will effect a permutation of data elements indirectly.