Practical Memory Checkers for Stacks, Queues and Deques

Marc Fischlin

Fachbereich Mathematik (AG 7.2) / Informatik
Johann Wolfgang Goethe-Universität Frankfurt am Main
PSF 111932
60054 Frankfurt/Main, Germany

e-mail: marc@informatik.uni-frankfurt.de
URL: http://www.uni-frankfurt.de/~roessner/group/marc/marc.html

Abstract. A memory checker for a data structure provides a method to check that the output of the data structure operations is consistent with the input even if the data is stored on some insecure medium. In [8] we present a general solution for all data structures that are based on insert(i, v) and delete(j) commands. In particular this includes stacks, queues, deques (double-ended queues) and lists. Here, we describe more time and space efficient solutions for stacks, queues and deques. Each algorithm takes only a single function evaluation of a pseudorandomlike function like DES or a collision-free hash function like MD5 or SHA for each push/pop resp. enqueue/dequeue command making our methods applicable to smart cards.

1 Introduction

A memory checker guarantees that the data that is retrieved from memory is consistent with the data inserted before. Consider for example a stack and the sequence push(a), push(b), pop, push(a), push(c), pop, pop. Then the expected output is $-$, $-$, b, $-$, $-$, c, a, where $-$ denotes "no output". Assume that the elements are kept on some insecure memory. Then an adversary, e.g. a virus, might tamper the content of the stack and thus produce a wrong output. Another source for errors can be a buggy program implementing the data structure. Using a memory checker, one will detect errors with high probability, regardless whether the errors are malicious or accidently.

We give some applications of memory checkers. Assume that a bank customer withdraws money from an automatic teller machine using his smart card. The bank keeps track of the operations by enqueuing every transaction, while the smart card contains only a small hash value for the memory checker. At the end of a month, the customer prints this sequence of transactions at the machine (using the smart card). Then the memory checker gives a fast method to verify that the output is correct and no further unauthorized transactions have been made.

In the previous example, all operations are enqueued before they are dequeued again. Consider a job queue on a computer system, e.g. the printer schedule queue. Here, insertions and deletions may alternate. Suppose that a clever user has found a possibility to manipulate the job queue and to make his jobs to be processed before others — or even to cancel other jobs. If the system runs a memory checker for queues (in a secure part), it can detect such errors.

Note that error detection with a memory checker is one-sided in the following sense: If the bank customer detects some error, he cannot use the faulty output protocol to accuse the bank of fraud, because he could have produced such a protocol himself. So the checker model can only be used to detect errors and to perform some countermeasures for the future.

At first glance, cryptographic signature or authentication schemes seem to be sufficient for designing memory checkers. Nevertheless, given a signature or message authentication code (MAC) z for the current content $(v_1, \ldots, v_n)$ of a queue and some element v that shall be enqueued, in some settings the cryptographic scheme must be able to compute the signature (or MAC) for $(v_1, \ldots, v_n, v)$ from z and v without accessing $v_1, \ldots, v_n$. Consider for example the data structure queue and the cipher block chaining mode of DES [1,12], i.e. the message authentication code for $x_1, \ldots, x_n \in \{0,1\}^{64}$ is

$$\text{CBC-MAC}_a(x_1, \ldots, x_n) := \begin{cases} \text{DES}_a(x_1) & \text{if } n = 1 \\ \text{DES}_a\left(\text{CBC-MAC}_a(x_1, ..., x_{n-1}) \oplus x_n\right) & \text{else} \end{cases}$$

Here, DES_a denotes the DES function with secret key a and $x \oplus y$ denotes the bitwise exclusive-or of x and y. Inserting an element at one end of the queue and deleting at the other isn't possible without reading all elements inserted so far. Even if it were possible, this would cause a large overhead.

Related work. The memory checker model was introduced by Blum et al. in [6] and refined in [8]. Blum et al. present checkers for stacks and queues, but their solution is based on ϵ-biased hash functions (see [11] for a definition). We use their ideas, but we don't exploit special properties of the underlying function family. That is, one can use *any* family inluding practical families like DES. In particular, the small and fast memory checker algorithm can be easily added to a smart card where algorithms for such a family are already implemented.

Recently, so called *incremental schemes* were introduced by Bellare, Goldreich and Goldwasser [2,3]. Informally, an $\mathcal{M}$-incremental signature or authentication scheme allows to produce a signature (or MAC) to some document M very fast, given a signature to document M' where M is obtained by applying a text modification in $\mathcal{M}$ to M'. The similarity to memory checkers is obvious. Hence, we derive efficient memory checkers if an $\mathcal{M}$-incremental scheme fullfils the following conditions:

- The modifications correspond to the data structure operations, e.g. for stacks we have text modifications "append-at-the-end" and "delete-from-the-end".
- As explained above, to update a signature the scheme merely needs the element that is inserted or deleted.

- The scheme is secure against so called *message substitution attacks*, i.e. one cannot produce a forgery even if one is allowed to tamper the message before an update step.

Incremental authentication schemes that support single block insertion and deletion were given in [3] and a more space and time efficient one in [8]. The latter one fullfils the abovementoned properties, but it produces about $n \log n$ bits authentication code, where n is the number of elements.

Exact security. We follow the paradigm of [5] presenting results in terms of exact security. The notion of exact security can be roughly described as follows: Let A be an adversary for the checker model with parameters t (running time[1]), q (number of data operations) and ϵ (success probability). From A we derive a distinguisher with running time t', number of oracle queries q' and success probability ϵ' for the function family used by the checker. Here, t', q', ϵ' are determined by t, q, ϵ. So, if you consider the function family to be (t', q', ϵ')-secure, you can immediately derive the checker's security level exactly.

2 Notations and Definitions

First, we define *function families.* Let F be a set of functions with the same domain $\{0,1\}^l$ and the same range $\{0,1\}^L$. Each function $f \in F$ is associated to a key. We write f_a for the function f specified by key a. Choosing a function f from F at random means choosing at random with equal probability a key a from the set of all keys and setting $f := f_a$. For example, the DES family consists of the DES function with input and output length 64, where each functions is specified by some 56 bit DES key. Given a function f with domain $\{0,1\}^l$ and n strings $x_1, \dots, x_n$ we sometimes write $f(x_1, \dots, x_n)$ instead of $f(x_1 \cdot x_2 \cdots x_n)$.

Let $\mathrm{Map}(X, Y)$ denote the family of all functions mapping X to Y where the key describing a function is the concatenation of all $|X|$ function values in some fixed order. Let $F \subseteq \mathrm{Map}(X, Y)$ be a function family. Informally, a family F is pseudorandom, if a random function $f \in F$ behaves almost like a true random function and $f(x)$ can be evaluated fast for all $x \in X$. It is widely believed that DES has this property. More formally, let $F \subseteq \mathrm{Map}(X, Y)$ be a function family and D a probabilistic algorithm. We write $D^f \in \{0,1\}$ for the output of D with oracle access to a function $f \in F$. Given two families $F, G \subseteq \mathrm{Map}(X, Y)$, the *advantage* of algorithm D distinguishing F and G is defined as

$$\mathrm{Adv}_D(F, G) = \mathrm{Prob}_{f \in F}\left[D^f = 1\right] - \mathrm{Prob}_{g \in G}\left[D^g = 1\right],$$

where the probabilities are taken over the internal coin tosses of D and the random choice of $f \in F$ resp. $g \in G$. Algorithm D is called a (t, q, ϵ)-*distinguisher* if it makes at most t steps in the standard RAM model, makes at most q oracle

[1] Formally, t includes the running time *and* the description size of the adversary's algorithm. For simplicity, in this paper we only consider the running time (measured in terms of number of RAM steps).

queries and achieves $\mathrm{Adv}_D\left(F, \mathrm{Map}(X,Y)\right) \geq \epsilon$. A family is called (t,q,ϵ)-*secure* if there is no (t,q,ϵ)-distinguisher for this family.

Finally, we give an informal definition of collision-free hash functions. See [7] for a formal treatment. A function $h : \{0,1\}^b \to \{0,1\}^k$ with $b > k+1$ is a *collision-free hash function*, if it is infeasible to find $x \neq x'$ such that $h(x) = h(x')$. Candidates in practice include MD5 and SHA, where $b = 512$ and $k = 128$ resp. $k = 160$. It is well known [7,10] that given a collision free hash function h, one can easily derive a collision-free iterated hash function $H : B^* \to \{0,1\}^k$ for $B := \{0,1\}^{b-k-1}$. Namely, let $x_1, \ldots, x_n \in B$ and set

$$H(x_1, \ldots, x_n) := \begin{cases} h(0^k, 0, x_1) & \text{if } n = 1 \\ h(H(x_1, \ldots, x_{n-1}), 1, x_n) & \text{else} \end{cases}$$

Finding a collision for H implies finding a collision for h.

3 Memory Checkers

We briefly review the definition of a memory checker [6,8]. A memory checker filters the interaction between a user and a data structure storing the data to some insecure memory. See figure 1. An execution is divided into rounds. At the beginning of each round, the checker gets the next *user operation*, e.g. a $\mathsf{push}(v)$ command, and performs some local computation, including arbitrary interaction with the data structure. Our checkers presented below only compute a function value, increment and decrement counters and pass the operation (perhaps adding some time stamp) to the data structure. At the end of the round, the checker shall return the output to the user *before* reading the next operation (or return "–" if the operation doesn't produce an output). The execution ends, when there are no more input operations left and the checker has finished its "postprocessing". If some error occurs the checker shall output BUGGY with high probability during or immediately after the execution. Conversely, if the execution is correct the checker shall never output BUGGY.

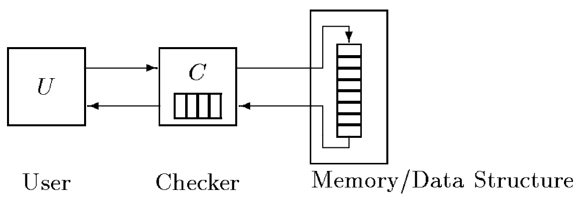

Fig. 1. The memory checker model

We stress that the local memory of the checker cannot be read or tampered neither by the user nor the adversary. From a theoretical point of view, our checkers remain valid if we allow the adversary to read the checker's memory except for the secret key specifying the function. Though in practice, in this case a DES checker producing a 64 bit check code might be for example vulnerable to birthday attacks.

To capture the worst case for the checker, we assume that the adversary totally controls the insecure memory and the input/output behaviour of the data structure. Furthermore, he controls the user and therefore chooses the user operations passed to the checker. The adversary works adaptively, i.e. he bases his decision on the previous steps of the protocol. A checker is called *on-line* if it detects an error immediately after it occured, i.e. before reading the next input operation. Otherwise it is called *off-line*. All our checkers presented below are off-line. Furthermore, a checker is called *noninvasive* if the insecure memory contains only values specified by the user operations (assuming that the adversary doesn't tamper this content and that the data structure works correctly). Else it is called *invasive*. Our checkers for stacks and deques are invasive as they add some time stamp, while our checkers for queues are noninvasive.

Definition 1. A memory checker C for a data structure $\mathcal{D}$ is (t, q, ϵ)-secure iff for every adversary A making at most t steps and passing at most q user operations to C, the probability that A returns a wrong value and C doesn't output BUGGY is less than ϵ. Additionally, the checker never outputs BUGGY if the output is correct for all operations.

We assume that all values are from $\{0,1\}^n$ and for stacks and deques we suppose that these data structures are capable of storing pairs from $\{0,1\}^n \times \{0,1\}^N$. Our checker will use the N extra bits to append a time stamp to every value. The inital configurations of stacks, queues and deques are empty.

4 Checkers for Stacks, Queues and Deques based on DES

In this section we define checkers for stacks, queues and deques, i.e. queues that allow to enqueue and dequeue at both sides. The checkers are all based on pseudorandomlike functions like DES. If the value length exceeds 64 bit DES input length, one can use for example the CBC construction [5] to stretch the input length to some multiple of 64 bits.

For the rest of this section, let F be an arbitrary family of functions mapping $\{0,1\}^l$ to $\{0,1\}^L$ with key length k.

4.1 A Checker for Stacks

We define an off-line checker for stacks. The private memory of the checker contains the following values:

- an N-bit time counter s, which is initialized with 0
- an N-bit position counter p, which is initialized with 0
- a k-bit key a of a function family F specifying a function f_a in F

– an L-bit value τ, which is initialized with 0^L

The key a is chosen at random in a preprocessing phase. The code τ will be updated every time an element is pushed or popped. If all retrieved values are correct, we will have $\tau = 0^L$ at the end of the execution.

For an user command push(v) the checker works as follows: It pushes the values (v, s) to the memory, computes $\tau := \tau \oplus f_a(v, p, s)$ and increments s and p. For a pop command, the checker pops a pair (v, s_v) from the memory and verifies that $s > s_v$. If not, it outputs BUGGY. Otherwise it passes v to the user, decrements p and computes $\tau := \tau \oplus f_a(v, p, s_v)$.

As the checker maintains a counter p for the number of elements in the stack, we may presume wlog. that the adversary never outputs "empty" unless the stack really is. If no more user operations are left, the checker empties the memory in a verification phase. It pops all elements and proceeds as above (without passing values to the user), until $p = 0$ and the stack is empty. We'll discuss the practical consequences of this overhead caused by the verification phase in section 6.

Note that the checker is deterministic — except for the random choice of the key a. The following lemma states that for $F = \mathcal{R} = \mathrm{Map}(\{0,1\}^l, \{0,1\}^L)$, errors will be detected with high probability:

Lemma 2. *Let f_a be a random function in $\mathcal{R} = Map(\{0,1\}^l, \{0,1\}^L)$, $l \geq n + 2N$. If an error occurs, we have $\tau = 0^L$ at the end of the execution with probability 2^{-L}, where the probability is taken over the random choice of f_a and the coin tosses of the adversary. If every output is correct, the final value of τ will be 0^L for any function family $F \subseteq Map(\{0,1\}^l, \{0,1\}^L)$.*

Proof. If no error occurs, every pushed value (v, s) is retrieved from the memory again (at the same position), so xoring these function values equals 0^L. In this case we have $\tau = 0^L$ at the end of the execution regardless of the function family.

Assume that an error occurs. We sort the sequence of push and pop commands according to the position. Namely, let $v_{j,i}$ be the value that the i^{th} push command for position j inserts. Let $w_{j,i}$ the value that is returned the next time a value is read from position j. Let the corresponding time values be $s_{v_{j,i}}$ resp. $s_{w_{j,i}}$. Then for every position j we have a sequence of m_j pairs, on which the function f_a is evaluated:

$$(v_{j,1}, j, s_{v_{j,1}}), (w_{j,1}, j, s_{w_{j,1}}), \ \ldots, \ (v_{j,m_j}, j, s_{v_{j,m_j}}), (w_{j,m_j}, j, s_{w_{j,m_j}})$$

Since an error occured, there exists i, j such that $(v_{j,i}, j, s_{v_{j,i}}) \neq (w_{j,i}, j, s_{w_{j,i}})$. We show that there must be a tuple that appears an odd number of times (or the checker detects an error immediately).

As the time stamps $s_{v_{j,x}}$ for $x = 1, \ldots, m_j$ are in increasing order, we have $(v_{j,x}, j, s_{v_{j,x}}) \neq (v_{j,y}, j, s_{v_{j,y}})$ for all $x < y$. Hence, the triples $(v_{j,x}, j, s_{v_{j,x}})$ can only appear for an even number of times, if there's a permutation π over $\{1, \ldots, m_j\}$ implying a bijection between $\{(v_{j,x}, j, s_{v_{j,x}}) \mid x = 1, ..., m_j\}$ and $\{(w_{j,y}, j, s_{w_{j,y}}) \mid y = \pi(1), \ldots, \pi(m_j)\}$. Assume for contradiction that $x > y := \pi(x)$ for some x. Then $(v_{j,x}, j, s_{v_{j,x}}) = (w_{j,y}, j, s_{w_{j,y}})$. As the checker verified that $s_{w_{j,y}}$ was less than the current counter s at the time $w_{j,y}$ was returned, we derive the contradiction $s_{v_{j,x}} \geq s > s_{w_{j,y}}$. Hence, $\pi(x) = x$ for all x, but this is a contradiction to our assumption $(v_{j,i}, j, s_{v_{j,i}}) \neq (w_{j,i}, j, s_{w_{j,i}})$. Thus, there exists some tuple that appears an odd number of times.

Let $M = \{(v_1, p_1, s_1), \ldots, (v_m, p_m, s_m)\}$ be the set of all triples that appear an odd number of times. Since there is such a triple, we have $m \geq 1$. At the end of the execution, τ equals $0^L \oplus f_a(v_1, p_1, s_1) \oplus \cdots \oplus f_a(v_m, p_m, s_m)$. The probability that

$$f_a(v_m, p_m, s_m) = f_a(v_1, p_m, s_1) \oplus \cdots \oplus f_a(v_{m-1}, p_{m-1}, s_{m-1})$$

and therefore $\tau = 0^L$ is equal to the probability that some random value from $\{0,1\}^L$ equals a fixed element in $\{0,1\}^L$. This probability is 2^{-L}. □

Theorem 3. *Let $F \subseteq Map(\{0,1\}^l, \{0,1\}^L)$, $l \geq n + 2N$, be a function family which is (t, q, ϵ)-secure. Then the checker for stacks described above is (t', q', ϵ')-secure, where $t' = t - cq$, $q' = \frac{1}{2}q$ and $\epsilon' = \epsilon + 2^{-L}$ for some small constant $c \in \mathbb{N}$.*

Suppose that we use DES. As cq will be small in comparison to t, we have $t \approx t'$. Additionally, q' and q differ only by a factor 2 and $2^{-L} = 2^{-64}$ is almost zero. Thus, the checker is roughly speaking as secure as DES.

Proof. Assume that there is an adversary A with running time t' that passes at most q' user operations to C and achieves success probability at least ϵ'. From A we construct a distinguisher D for F and $\mathcal{R} = \text{Map}(\{0,1\}^l, \{0,1\}^L)$ with running time t, making at most q oracle queries and advantage at least ϵ. Wlog. we presume that A never returns a timer value that is equal to or greater than the timer value of the checker — since such errors will be detected immediately. Furthermore, we assume wlog. that A never returns "empty" though the stack isn't (because the checker maintains the number of elements in the stack), but returns at least one wrong value during the execution.

D is given oracle access to a random function g in F or $\mathcal{R}$. It performs a black-box-simulation of A by running the checkers program using the oracle access to g. That is, it maintains two counters s, p (initialized with 0) and a code τ (initialized with 0^L). Whenever the checker would have updated τ by xoring it with $f_a(v, p, s)$, D computes $\tau = \tau \oplus g(v, p, s)$ and updates p, s accordingly. At the end of the execution, D outputs 1 iff $\tau = 0^L$. Therefore,

$$\begin{aligned}
\text{Adv}_D(F, \mathcal{R}) &= \text{Prob}_{g \in F}\left[D^g \text{ outputs } 1\right] - \text{Prob}_{g \in \mathcal{R}}\left[D^g \text{ outputs } 1\right] \\
&= \text{Prob}_{g \in F}\left[\tau = 0^L \text{ at the end}\right] - \text{Prob}_{g \in \mathcal{R}}\left[\tau = 0^L \text{ at the end}\right] \\
&= \text{Prob}_{g \in F}\left[A \text{ is successful}\right] - \text{Prob}_{g \in \mathcal{R}}\left[A \text{ is successful}\right] \\
&\geq \epsilon' - 2^{-L}
\end{aligned}$$

The running time of D equals the running time of A plus the time to increment and decremement the counters resp. to compute the new τ in every step. As the checker empties the stack after having answered the last user operation, D needs at most q' additional oracle queries. □

4.2 A Checker for Queues

Our checker for queues is noninvasive, i.e. it doesn't append a time stamp. The local memory of the checker contains two N-bit position counters p_{top} and p_{bot} (both initialized with 0), a random k-bit key a specifying a function f_a in F and an L-bit code τ initialized with 0^L. Whenever we enqueue a value, we increment p_{top}. If we dequeue a value, we increment p_{bot}. Thus, $p_{\text{top}} - p_{\text{bot}}$ is the number of elements currently in the queue To be more precise, if the checker shall enqueue an element v, it enqueues v, computes $\tau := \tau \oplus f_a(v, p_{\text{top}})$ and increments p_{top}. If the checker shall dequeue a value, it dequeues v, returns v to the user, computes $\tau := \tau \oplus f_a(v, p_{\text{bot}})$ and increments p_{bot}. At the end, the checker dequeues all elements in the verification phase updating p_{bot} and τ as above. Security is proven as in Lemma 2 and Theorem 3.

Lemma 4. *Let f_a be a random function in $\mathcal{R} = Map(\{0,1\}^l, \{0,1\}^L)$, $l \geq n+N$. If an error occurs, we have $\tau = 0^L$ at the end of the execution with probability 2^{-L}, where the probability is taken over the random choice of f_a and the coin tosses of the adversary. If no error occurs, at the end $\tau = 0^L$ holds for any function family $F \subseteq Map(\{0,1\}^l, \{0,1\}^L)$.*

Proof. At the end of the execution, the function has been evaluated for $2m = 2(p_{\text{top}} - 1)$ tuples (v_j, j), (w_j, j), $j = 1, \ldots, m$, where v_j is the j^{th} enqueued element and w_j is the j^{th} value dequeued. If $v_j = w_j$ for all j we obviously have $\tau = 0^L$ at the end. If $v_j \neq w_j$ for some j we derive that $\tau \neq 0^L$ with probability 2^{-L} as in Lemma 2. □

Theorem 5. *Let $F \subseteq Map(\{0,1\}^l, \{0,1\}^L)$, $l \geq n+N$, be a function family that is (t, q, ϵ)-secure. Then the checker for queues described above is (t', q', ϵ')-secure, where $t' = t - cq$, $q' = \frac{1}{2}q$ and $\epsilon' = \epsilon + 2^{-L}$ for some small constant $c \in \mathbb{N}$.*

4.3 A Checker for Deques

To derive a checker for deques, we introduce time stamps again, as stacks can be viewed as special deques. Let $\mathsf{enqueue}^L$, $\mathsf{enqueue}^R$, $\mathsf{dequeue}^L$ and $\mathsf{dequeue}^R$ be the commands to enqueue resp. dequeue elements at the left or right side. Let F be function family with input length $l \geq n+2N+1$. The checker maintains five N-bit counter values s, p^L_{top}, p^R_{top}, p^L_{bot} and p^R_{bot}, all initialized with 0. Additionally, it initializes τ with 0^L. If the checker gets an $\mathsf{enqueue}^L(v)$ or $\mathsf{enqueue}^R(v)$ command, it computes $\tau := \tau \oplus f_a(0, v, p^L_{\text{top}}, s)$ (for $\mathsf{enqueue}^L$) or $\tau := \tau \oplus f_a(1, v, p^R_{\text{top}}, s)$ (for $\mathsf{enqueue}^R$), enqueues the pair (v, s) at the corresponding side and increments s and p^L_{top} resp. p^R_{top}.

To process a $\mathsf{dequeue}^L$ command, the checker works as follows: If $p^L_{\text{top}} > p^L_{\text{bot}}$ it dequeues a pair (v, s_v). If $s_v \geq s$, it outputs BUGGY. Otherwise, it decrements p^L_{top} and sets $\tau := \tau \oplus f_a(0, v, p^L_{\text{top}}, s_v)$. Now assume that $p^L_{\text{top}} = p^L_{\text{bot}}$ (the case $p^L_{\text{top}} < p^L_{\text{bot}}$ will never occur). If $p^R_{\text{top}} = p^R_{\text{bot}}$ the deque is empty and the checker outputs "empty". Else $p^R_{\text{top}} > p^R_{\text{bot}}$, and we dequeue a pair (v, s_v), output BUGGY if $s_v \geq s$ and otherwise compute $\tau := \tau \oplus f_a(1, v, p^R_{\text{bot}}, s_v)$ and increment p^R_{bot}. Note that though we dequeue from left, we update τ as we would dequeue from the right side. A $\mathsf{dequeue}^R$ command is processed similar with roles of left and right flipped.

At the end of the execution the checker dequeues elements in a verification phase via $\mathsf{dequeue}^L$ commands as long as $(p^L_{\text{top}} - p^L_{\text{bot}}) + (p^R_{\text{top}} - p^R_{\text{bot}}) > 0$. Security follows as before, since arguments for the function f_a are prepended by 0 or 1 depending on the side on which the elements are enqueued or dequeued. If all values are correct, we'll have $\tau = 0^L$, while an error will be detected with high probability. The proof of the following Theorem is similar to the proofs of Lemma 2 and Theorem 3.

Theorem 6. *Let $F \subseteq Map(\{0,1\}^l, \{0,1\}^L)$, $l \geq n+2N+1$, be a function family that is (t, q, ϵ)-secure. Then the checker for deques described above ist (t', q', ϵ')-secure, where $t' = t - cq$, $q' = \frac{1}{2}q$ and $\epsilon' = \epsilon + 2^{-L}$ for some small constant $c \in \mathbb{N}$.*

5 Checkers for Queues based on Iterated Hash Functions

In this section, we present a simple checker based on iterated hash functions defined in section 2. Recall that $H : B^* \to \{0,1\}^k$ for $B := \{0,1\}^{b-k-1}$ for the underlying hash function $h : \{0,1\}^b \to \{0,1\}^k$. For notational convenience let $b^* = b - k - 1$. Our checker doesn't need a position or time counter, nor does it append a time stamp to each value. Unfortunately, security cannot be stated in terms of exact security until we use families of hash functions. A formal treatment based on the results of [7] will be given in the final version.

To check a queue we maintain two hash values $e, d \in \{0,1\}^k$, both initialized with $h(0^k, 0, 0^{b^*})$. To enqueue a value $v \in B$, let $e := h(e, 1, v)$. To dequeue a value v, compute $d := h(d, 1, v)$ and pass v to the user. If the data structures claims that the instance is empty, we verify that $e = d$ and output BUGGY if not. If $e = d$ we return "empty" to the user. If no more operations are left, we dequeue all elements in a verification phase until the data structure returns "empty". In this phase, we update d for each element as above without returning values to the user. Finally, the checker verifies that $e = d$ and outputs BUGGY if not.

We always have $e = H(0^{b^*}, v_1, \dots, v_n)$ for the sequence $v_1, \dots, v_n \in B$ of elements enqueued. Additionally, it always holds $d = H(0^{b^*}, w_1, \dots, w_m)$ for the sequence $w_1, \dots, w_m \in B$ of elements dequeued.

Proposition 7. *The checker presented above is secure unless one can find a collision for the hash function h.*

Proof. Let $v_1, \dots, v_n$ resp. $w_1, \dots, w_m$ be the sequence of enqueued resp. dequeued elements. If no error occurs, we have $n = m$ and $v_i = w_i$ for all i, since queues are FIFO systems. Hence, $e = d$ at the end of the execution. Assume that an adversary fools the checker. There are two possibilities: First, at some point the adversary outputs "empty" though the queue isn't. In this case, the checker immediately verifies that $e = d$. As the adversary is successful, we have $e = H(0^{b^*}, v_1, \ldots, v_{n'}) = H(0^{b^*}, w_1, \ldots, w_{m'}) = d$ for $m' < n'$ and the adversary has found a collision for H. On the other hand, if the adversary never lies about the number of elements, we have $n = m$ and there exists some i such that $w_i \neq v_i$. Then $e = H(0^{b^*}, v_1, \ldots, v_n) = H(0^{b^*}, w_1, \ldots, w_n) = d$, and the adversary has found a collision for different messages of the same length. $\square$

6 How to proceed in practice

Suppose that we want to implement a checker on a multi user computer system like Unix. All the checkers presented above are off-line checkers. To verify the correctness of the memory content the checker must empty the instance. However, in most settings the data mustn't be deleted because we need it for further use.

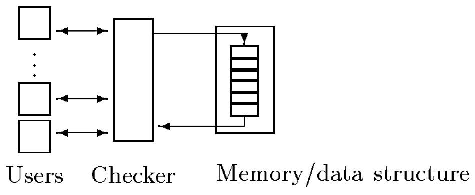

Fig. 2. A memory checker for queues

We suggest the following solution: The checker is maintained by the system administrator, and the users can access the common data structure only via the checker (see figure 2) — assuming that no read/write conflicts occur. Whenever the access frequency drops, e.g. at night, the checker temporarily forbids any operation and performs a check. Consider for example stacks. To verify, the checker empties one instance and immediately inserts each element again in another instance. To be more precise, let τ be the current check value for that instance. The checker initializes new counter values p' with 0 and s' with s and sets $\tau' = 0^L$. Then we pop the first element from the old stack (updating s, p, τ as above), and insert it again in the new instance (this time updating p', s' and τ' instead of p, s, τ). We repeat this for the other elements. When all elements are deleted from the old instance, the checker verifies that $\tau = 0^L$ — and outputs BUGGY if not. Note that all values in the new instance are in reverse order, so we do the same again swapping the roles of the old and new instance resp. p, s, τ and p', s', τ' to get the correct order again. For queues and deques the verification phase is done similar though we don't have to reverse the order again.

Refreshing the key for DES based checkers can be done in the verification phase. At the beginning of such a phase, we choose a new random DES key a' and reset s' to 0 (instead of setting $s' = s$). Then we work as described above but we compute τ' via the function $f_{a'}$ specified by a' though τ is still updated using f_a. For stacks the key refreshment may be done in the reverse order phase.

Our experimental results (Appendix A) show that the overhead caused by the checker for each user operation is insignificant. In contrast to that, the verification phase of an off-line checker is very expensive — if there are many elements in the instance. But as the examples of the bank customer and the printer schedule queue show, in some settings it is very likely that the user empties the instance anyway, making the verification phase fast. In particular, for smart cards it is preferable that (if possible) the underlying data structure is a queue and that the checker is based on iterated hash functions. The lower bounds for on-line checkers given in [6] and the algorithms for on-line stack checkers in that work indicate that we probably cannot find checkers that work on-line, have small private memory and do not store much extra data on the insecure memory. We leave it as an open problem to prove this conjecture or to develop such checkers.

References

1. ANSI X3.106: American National Standard for Information Systems — Data Encryption Algorithm — Modes of operation. American National Standards Institute, 1983.
2. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: The case of hashing and signing. In Crypto'94 (1994), vol. 839 of Lecture Notes in Computer Science, Springer-Verlag, pp. 216–233.
3. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography and application to virus protection. In Proceedings of the 27th Annual Symposium on the Theory of Computing (1995), pp. 45–56.
4. Bellare, M., Guérin, R., Rogaway, P.: XOR MACs: New methods for message authentication using finite pseudorandom functions. In Crypto'95 (1995), vol. 963 of Lecture Notes in Computer Science, Springer-Verlag, pp. 15–29.
5. Bellare, M., Kilian, J., Rogaway, P.: On the security of cipher block chaining. In Crypto'94 (1994), vol. 839 of Lecture Notes in Computer Science, Springer-Verlag, pp. 341–358.
6. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica **12** (1994), pp. 255–244.
7. Damgård, I.: A design priciple for hash functions. In Crypto'89 (1989), vol. 435 of Lecture Notes in Computer Science, Springer-Verlag, pp. 416–427.
8. Fischlin, M.: Incremental cryptography and memory checkers. In Eurocrypt'97 (1997), Lecture Notes in Computer Science, Springer-Verlag.
9. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of ACM **33(4)** (1986), pp. 792–807.
10. Merkle, R.: One way hash functions and DES. In Crypto'89 (1989), vol. 435 of Lecture Notes in Computer Science, Springer-Verlag, pp. 428–446.
11. Naor, J., Naor, M.: Small-bias probability spaces: Efficient constructions and applications. Journal on Computing **22** (1993), pp. 838–856.
12. National Bureau of Standards: Data Encryption Standard, Federal Information Processing Standard, Publication 46. US Department of Commerce, 1977.
13. National Bureau of Standards: Secure Hash Standard, Federal Information Processing Standard, Publication 180. US Department of Commerce, 1993.

14. Rivest, R.: The MD5 message-digest algorithm. IETF Network Working Group, RFC 1321, 1992.

A Experimental Results

In this section we present some experimental results for stacks with DES based checkers and for queues with SHA-1 based checkers. The algorithms were implemented in ANSI-C using the cryptlib package (available at site `http://www.cs.auckland.ac.nzl/~pgut001/cryptlib/index.html`) on a Linux system running on a DOS computer with Intel Pentium 166 chip. For the stack checker we use the CBC construction of DES to stretch the input length [5], allowing different element sizes with 128, 512, 1024, 2048 bit. The counter values are each restricted to 32 bit. For a fixed element size, the experiments showed that the running time grows proportional to the number of elements. Furthermore, repeating the exeperiment didn't change the time significantly. Thus, we only give the average results for 100, 000 elements.

element size	128 bit	512 bit	1024 bit	2048 bit
10^5 eval. of CBC-DES	2.8 sec	7.2 sec	13.0 sec	24.8 sec
10^5 stack commands	0.5 sec	1.9 sec	3.7 sec	7.4 sec
10^5 insertions	4.1 sec	10.8 sec	19.9 sec	38.3 sec
Verification phase	15.8 sec	40.9 sec	80.1 sec	156.5 sec

The first row shows how fast we can evaluate a CBC-DES on the system. We've applied the CBC-DES function 10^5 times for the zero string (of corresponding size) without processing the function output. The second row shows how fast we can do 10^5 stack commands, of which a half are push resp. pop commands. For simplicity, we've chosen an array representation of the data structure stack supporting a push and pop function. The following rows present the running time of the checker. The first one shows the time to insert 100, 000 elements. Note that this time is greater than the sum of the two proceeding rows, since we also have to maintain two counter values and must process the DES output. As expected, the running time of the verification phase is about four times the time to insert the elements. This follows from the fact that we have to pop every element, push it in another instance and do the same again to reverse the order.

For queues, we've used the construction based on iterated hash function with SHA-1. For simplicity, we've used 344 bit values, prepending it with the previous hash value and 8 bit representing the 0 or 1 bit as a character, such that the total length adds up to 512 bits. Again, we've chosen an array implementation of the data structure.

10^5 eval. of SHA-1	1.9 sec
10^5 enqueue/dequeue	1.8 sec
10^5 insertions	3.8 sec
Verification phase	7.8 sec

As the table confirms, the checker is much faster than a DES based one.