Stack and Queue Integrity on Hostile Platforms

Premkumar T. Devanbu
Department of Computer Science
University of California
Davis, California CA 95616 USA
devanbu@cs.ucdavis.edu

Stuart G. Stubblebine
AT&T Laboratories—Research
Florham Park, NJ 07932 USA
stubblebine@research.att.com

Abstract

When computationally intensive tasks have to be carried out on trusted, but limited, platforms such as smart cards, it becomes necessary to compensate for the limited resources (memory, CPU speed) by off-loading implementations of data structures on to an available (but insecure, untrusted) fast co-processor. However, data structures such as stacks, queues, RAMS, and hash tables can be corrupted (and made to behave incorrectly) by a potentially hostile implementation platform or by an adversary knowing or choosing data structure operations. This paper examines approaches that can detect violations of datastructure invariants, while placing limited demands on the resources of the secure computing platform.

1 Introduction

Smart cards, set-top boxes, consumer electronics and other forms of trusted hardware [2, 3, 16] have been available (or are being proposed [1]) for applications such as electronic commerce. We shall refer to these devices as $\mathcal{T}$. These devices are typically composed of a circuit card encased in epoxy or a similar substance, that has been strewn with various electronic tamper-detection devices. The physical design constraints on these devices include heat dissipation difficulties, size (often a credit card or PCMCIA format), very low power consumption requirements, etc. This leads (particularly in the credit card format) to devices with exceedingly low data transfer rates, memory, and omputing resources. These resources are not a limitation in a few applications areas as such as cash cards, identity cards etc. However, for general purpose multi-application cards, these resource limitations are significant.

We have been exploring the use of trusted hardware in software engineering [7, 6]. (see Section 6). In this context, it becomes necessary store large amounts of data in the form of various data structures (stacks, queues, arrays, dynamic/static symbol tables, various types of trees, etc.). Such data structures cannot fit into the limited resources on the $\mathcal{T}$device. However, $\mathcal{T}$ devices are usually used in concert with a larger, more powerful (and presumably adverse) host computer ($\mathcal{H}$). Such data structures can be stored on $\mathcal{H}$. But how can their integrity be assured? Data structures have invariants; for each instance of a data structure, these invariants can be stored in a "digested" form as signatures within the $\mathcal{T}$device. Data structure operations are performed in conjunctions with modifications on the signatures to maintain a "digested" form of the invariants.

This approach differs from earlier work [4, 5]. Our protocols are much simpler. We use $O(1)$ memory in the trusted computer, and transfer only $O(1)$ amount of data for each push and pop operation in an on-line mode. Previous approaches used a $O(log(n))$ trusted memory and $O(log(n))$ data transfer for each operation (where n is the size of the stack or queue). However, previous work [4, 5] assumed extremely powerful adversaries (information theoretic bounds). We do follow in this line of work, but with quite different techniques that are applicable with computationally bounded adversaries.

This paper is organized as follows. In Section 2, we present goals, threats, and related work. Section 3 describes our protocol for stacks and an evaluation of it. In Section 4, we describe our protocol for queues and an evaluation of it. Section 5 describes previous work to handle random access memory and discusses the relationship of stacks and queues to this work. Section 7 describes extensions to our protocols. Section 6 describes applications using our protocols. Section 8 presents some concluding remarks.

2 Background

In this section we discuss the goals of the work, threats, and related work.

The goal of this work is to maintain integrity of stacks and queues maintained by a potentially hostile

platform. Also, we wish to do this in a manner that is extensible for other desired properties described in Section 7.

We need to make some assumptions about the adversary and the environment. So that we may focus our description on the security of the data structure application, we assume the channel between $\mathcal{T}$ and $\mathcal{H}$ is authenticated. We allow for an adversary that may learn information on this channel. We assume $\mathcal{H}$ is dishonest and need not follow the protocol. We assume the adversary can also submit high level commands to $\mathcal{T}$. Thus, the data structure protocols need to be secure against chosen and known attacks. In this context, a *chosen attack* is one where an adversary has complete control over data and operations to the data structure. A *known attack* is one where the attacker is assumed to know operations and data. We assume a computationally bounded adversary who is limited in the number of operations that can be submitted to the data structure, the amount of information that may be stored, and the number of operations required to process data and/or fill storage with data.

We assume the adversary may try to replay data from other instances of data structures. These replays may be due to multiple concurrent runs of the protocol and should not lead to a vulnerability. We assume the adversary may try to compose new messages using message fragments from other sessions.

Some issues are beyond the scope of this paper. This paper does not address how to recover from corrupt or lost data. Thus, we do not attempt to replicate data structures and operations. The sharing of data structures by multiple secure processors is also beyond the scope of this paper.

2.1 Related Work

This work follows the memory protection investigations of [4, 5], which considered the problem of verifying the correctness of a large memory of size n bits maintained by an all-powerful adversary P, subject to update requests from originator V with only a limited amount of trusted memory. (Most of these schemes are based on Merkle signature trees [12] which is described in further detail in Section 5.) It is shown that P can fool V with an incorrect memory whenever V has access to less than $log(n)$ bits of trusted memory. They also describe implementations of stacks and queues [5]. The stack implementation uses $log(H)$ memory accesses for operations on a stack of height H. Our approach also relates to, but differs from the work of Lamport [10]. His one-time password scheme precomputes a chain of hashes on a secret w with the sequence: $w, H(w), H(H(w)), \ldots, H^t(w)$. The password fo the i^{th} idenfication session, $1 \leq i \leq t$, is defined to be $w_i = H^{t-i}(w)$. In Lamport's scheme, the chain decreases with each useage. We also use a chain of "digests" (signatures and/or hashes) in our protocols; however, our scheme computes the chain differently; in addition, our chain grows and shrinks based on the change in state of the data structure.

In our approach, we expect that P is a constant factor faster than the V. We use only a constant number of bits of trusted memory, irrespective of the size of the stack and the queue. We also perform only a constant number of untrusted memory operations for each stack push and pop. We assume a "signature scheme" that is collision/computation resistant and 2^{nd} preimage resistant.

Goldreich [8] and Ostrovsky [14] give solutions for oblivious machines. A machine is oblivious if the sequence in which it accesses memory locations is equivalent for any two programs with the same running time. This work solves a different problem yet relies on techniques for protecting the integrity of memory (e.g., Ostrovsky uses sequence numbers for protecting RAMs), but does not address methods for protecting the integrity of stacks and queues.

3 Stacks

We begin by defining a stack as follows:

Stack $\langle T \rangle$
Interface
    $push : Stack\langle T \rangle \times T \rightarrow Stack\langle T \rangle$
    $pop: Stack\langle T \rangle \rightarrow T$
    $new\langle T \rangle : unit \rightarrow Stack\langle T \rangle$
    $delete\langle T \rangle : Stack\langle T \rangle \rightarrow unit$
Invariants
    $pop(push(x, S)) = x$
    $pop(new()) =$ error

We propose to implement this using a secure processor $\mathcal{T}$ (for *trusted*), and an insecure processor $\mathcal{H}$ (for *hostile*), using the following algorithm. Actions taken by $\mathcal{T}$ are shown so prefixed, others in italics. $r_i's$ are random numbers, generated by $\mathcal{T}$. $\sigma(x)$ is a signature on datum x by the trusted processor. In the basic protocol (i.e., where there is only one trusted host), $\sigma(x)$ can represent a cryptographic hash function (either keyed or unkeyed) or a public key based digital signature. We assume $\sigma(x)$ is collision/computation resistant and 2^{nd} pre-image resistant.

We assume the probability of a signature collision can be made arbitrarily small by changing the parameters of the signature scheme. We also assume that

there is a good source of random numbers; such functionality is starting to become available from hardware devices. If $\sigma(x)$ is keyed, a separate key is generated for each instance. The key is destroyed upon a delete and the key never leaves $\mathcal{T}$.

We assume a authenticated channel with message stream integrity between $\mathcal{T}$ and $\mathcal{H}$. Also, entries on the stack can be simple strings. Arrows indicate direction of transmission. Below, we use the " $'$ " as in σ' to represent a new value for σ. Data structure operation requests from $\mathcal{T}$ to $\mathcal{H}$ are shown in double quotes (e.g., "push ...") followed by the relevant operands.

Protocol 1 (Stack Operation)

For new():

$\mathcal{T}$	*selects random* $r_{init} \in_R \{0,1\}^l$
$\mathcal{T} \rightarrow \mathcal{H}$	*"new stack", label:* $\sigma(r_{init})$
$\mathcal{H} \rightarrow \mathcal{T}$	*"Done"*
$\mathcal{T}$	$\sigma' \leftarrow \sigma(r_{init})$

To initialize a stack, the $\mathcal{T}$ generates a new random number, signs it and sends it off to $\mathcal{H}$ with the "new stack" command. Henceforth, this signature $\sigma(r_{init})$ is used by $\mathcal{T}$ to identify the stack.

For push(x, S):

$\mathcal{T} \rightarrow \mathcal{H}$	*"push stack",* x, σ*, label:* $\sigma(r_{init})$
$\mathcal{H}$	*(stores the above two on top of stack)*
$\mathcal{H} \rightarrow \mathcal{T}$	*"Done"*
$\mathcal{T}$	$\sigma' \leftarrow \sigma(x \mid\mid \sigma)$

Here, a "push" request is sent to $\mathcal{H}$ along with the current stack signature and the new value. The stack signature is updated by signing the string formed by appending the pushed value to the current signature. This signature is always retained in $\mathcal{T}$, and we refer to it below as σ_{top}.

For pop(S)

$\mathcal{T} \rightarrow \mathcal{H}$	*"pop stack", label:* $\sigma(r_{init})$
$\mathcal{H} \rightarrow \mathcal{T}$	σ_{top} *and* x *(from top of stack), or* $\sigma(r_{init})$ *and "error" if stack underflow.*
$\mathcal{T}$	*if not "error", computes* $\sigma(x \mid\mid \sigma_{top})$ *and compare it with stored* σ
$\mathcal{T}$	*if "error", compares* σ *with* $\sigma(r_{init})$*;*
$\mathcal{T}$	*if above comparisons fail, terminate with error*
$\mathcal{T}$	*otherwise* $\sigma' \leftarrow \sigma_t$

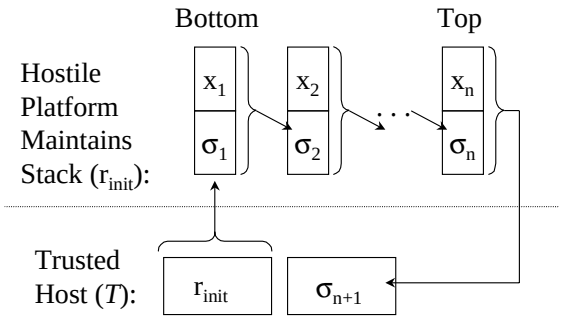

Figure 1: A resource-limited, secure implementation of stacks.

Upon a "pop", the $\mathcal{H}$ returns the value ostensibly at the top of the stack (x), and the signature of the stack (σ_{top}) when that value was pushed. The stored, current stack signature is recomputed and checked as shown above. $\mathcal{T}$ verifies an empty stack claim by recomputing $\sigma(r_{init})$ using the local copy of r_{init} and comparing it to the returned copy of σ.

For delete(S):

$\mathcal{T} \rightarrow \mathcal{H}$	*"delete stack", label:* $\sigma(r_{init})$
$\mathcal{H} \rightarrow \mathcal{T}$	*"Done"*
$\mathcal{T}$	*Removes* (σ, r_{init}) *tuple.*

The "delete" command resets the stack protocol; associated signatures held by $\mathcal{T}$ are discarded.

The stack protocol is illustrated in Figure 1. The arrows show the inputs to the computed signature. There is always a signature of the stack maintained in the $\mathcal{T}$ device. Prior to executing the push, the signature σ of the stack is in the $\mathcal{T}$ device; when an item x_i needs to be pushed on, as the i'th member of the stack, $\mathcal{T}$ computes a new signature σ_{i+1} as shown above and in the figure. Then the new item x and the old signature σ_i are given to the $\mathcal{H}$ stack implementation, with a request to execute a push. Normally, a push takes one argument, but since we are using a fixed length signature, these two arguments (the new item and the old signature) can just be represented as a single bit string. The inclusion of the signature adds a constant amount of external storage and transmission overhead for each operation. The new signature σ_{i+1} is retained in the $\mathcal{T}$ device's memory as defense against tampering by $\mathcal{H}$. Thus, when a pop command is issued, $\mathcal{H}$ is expected return the top item x, and signature of the rest of stack, σ_i. Then the original signature σ_{i+1} is recomputed and checked against the value stored in

$\mathcal{T}$. It is infeasible for $\mathcal{H}$ to spoof $\mathcal{T}$ by forging the values of x_i or σ_i so long as $\mathcal{T}$ retains σ_{i+1}. Thus the stack invariants are preserved. This is argued in more detail in the following section.

3.1 Evaluation

We now argue that our stack integrity checking protocols work provided the signature schemes we use are collision resistant and 2^{nd} pre-image resistant.

Definition 1 *An* ideal stack *is one which works correctly according to the standard specification of a stack.*

Specifications of a stack can be found on page 2, and in [9] (page 170). Non-ideal stacks only show their flaws when a pop is executed that returns a value other than what should be on the top of an ideal stack.

Definition 2 *We define an* incorrect stack *as one which after some series of operations* $\Omega = o_1, \ldots o_n, pop(stack_{id})$ *returns a value different from the one that would be on the top of an ideal stack after the operations* $o_1, \ldots o_n$.

Definition 3 *A protocol for checking stack implementations is secure if an incorrect stack is always detected whenever it returns the wrong value on a pop.*

We can now present the main claim of correctness of our stack protocol.

Theorem 1 *Our stack protocol is secure, as long as the signature scheme on which it is based is collision resistant and* 2^{nd} *pre-image resistant.*

To prove this theorem, we first define the notion of a *correct digest* of a stack. Next, we argue that our protocols ensure that $\mathcal{T}$, after any series of operations Ω will have a correct digest of the ideal stack after operations Ω. We then argue that if $\mathcal{T}$ has the correct digest of the ideal stack, then an incorrect stack operation by $\mathcal{H}$ will be detected.

Definition 4 *A correct digest of a stack with initializing value* s_0, *and items* $s_1, \ldots s_n$ *is defined as follows:*

$$\sigma_0 = \sigma(s_0)$$

$$\sigma_i = \sigma(s_i \mid\mid \sigma_{i-1}) \ , i = 1 \ldots n$$

We are now ready to state the main claim about the digest maintained by our protocol.

Claim 1 *After any series of operations* $\Omega = o_1 \ldots o_n$ *Our protocol always maintains the correct digest of an* ideal *stack in* $\mathcal{T}$, *providing a)* $\mathcal{T}$ *operates correctly according to our protocol, and b) the underlying signature scheme is collision resistant and* 2^{nd} *pre-image resistant.*

This is shown by induction, assuming that the $\mathcal{T}$ works according the stack protocol given above. The correctness of the digest initial state is trivial. Now suppose that we have a correct digest σ_{i-1} of the ideal stack after the first $i-1$ operations. There are two significant cases: o_i may be a push or a pop.

Push For $push(x)$, the $\mathcal{T}$ will compute a signature thus:

$$\sigma_i = \sigma(x \mid\mid \sigma_{i-1})$$

which is correct by definition (see description of "push" in Protocol 1) and by inductive assumption.

Pop Upon a pop, the $\mathcal{H}$ is expected to return two values: the item at the top of the stack x, and the signature of the rest of the stack σ_r. $\mathcal{T}$ checks that the following holds:

$$\sigma_{i-1} = \sigma(x \mid\mid \sigma_r)$$

Since we assume a collision resistant and 2^{nd} pre-image resistant signature scheme, it would be infeasible for $\mathcal{H}$ to find other values of x, σ_r so as to satisfy the above constraint. So x and σ_r are indeed the same values that were used to compute σ_{i-1} originally. This means that if σ_{i-1} correctly digests the ideal stack, then so does σ_r after the pop is executed.

The operation may also be a *delete* or a *new*; in either case, the effect will be either to create a new, independent, (correctly initialized) digest of a new stack, and/or to terminate the current stack instance (even if there are elements in it).

Our final claim is shown below; when this claim is established, the theorem is proven.

Claim 2 *If the* $\mathcal{T}$ *always stores the correct digest of the ideal stack, after every sequence of operations* $\Omega = o_1 \ldots o_n$, *then an incorrect stack operation can always be detected, provided the underlying signature scheme is collision resistant and* 2^{nd} *pre-image resistant.*

Without loss of generality, assume that after the operations Ω above, we execute a *pop*. Assume the $\mathcal{T}$ correctly digests the ideal stack after the operations Ω into σ_n, which was (during some operation $o_i, 1 \leq i \leq n$) computed as:

$$\sigma_n = \sigma(x \mid\mid \sigma_r)$$

where x is the item currently on the top of the stack. Now, because of the collision resistance of signature scheme, $\mathcal{H}$ cannot feasibly substitute another x or σ_r. Therefore, an incorrect stack operation will be detected via a bad signature.

4 Queues

Queues are implemented by keeping two items in trusted memory — the signature of the entire queue, including the items that used to be at the rear of the queue, and a signature of all the items that have been removed from the queue.

We begin with a brief description of the interface of a queue.

Queue $\langle T \rangle$
Interface

$nq : Queue\langle T\rangle \times T \rightarrow Queue\langle T\rangle$
dq: $Queue\langle T\rangle \rightarrow T$
$new\langle T\rangle : unit \rightarrow Queue\langle T\rangle$
$delete\langle T\rangle : Queue\langle T\rangle \rightarrow unit$

Axiomatization is not provided; it can be found in standard texts on formal specification, such as Guttag & Horning [9]. As in the case of stacks, we assume messages between $\mathcal{T}$ and $\mathcal{H}$ are sent over an authenticated channel having message stream integrity, bit-string entries on the queue, and (for simplicity) a new signing key for each queue instance.

Protocol 2 (Queue Operation)
For new(Q):

$\mathcal{T}$	*selects random* $r_{init} \in_R \{O,1\}^l$
$\mathcal{T} \rightarrow \mathcal{H}$	*"Init Queue" label* : $\sigma(r_{init})$
$\mathcal{H} \rightarrow \mathcal{T}$	*"Done"*
$\mathcal{T}$	$\sigma_q \leftarrow \sigma(r_{init}), \sigma_r \leftarrow \sigma(r_{init})$

As in the case of the stack, $\mathcal{T}$ generates a new random number, signs it, and sends it to $\mathcal{H}$ as an identifier to initialize a new queue.

For nq(Q, x):

$\mathcal{T} \rightarrow \mathcal{H}$	*"enqueue"* $x, \sigma(x \mid\mid \sigma_q)$, *label* : $\sigma(r_{init})$
$\mathcal{H} \rightarrow \mathcal{T}$	*"Done"*
$\mathcal{T}$	${\sigma'}_q \leftarrow \sigma(x \mid\mid \sigma_q)$

On an enqueue, $\mathcal{T}$ computes a new signature by signing the string formed by appending the new item with the current signature of the entire queue. This signature is sent to $\mathcal{H}$ along with the current item; this signature also updates the current queue signature after the enqueue operation.

For dq(Q):

$\mathcal{T} \rightarrow \mathcal{H}$	*"dequeue", label:* $\sigma(r_{init})$
$\mathcal{H} \rightarrow \mathcal{T}$	σ_{front}, x

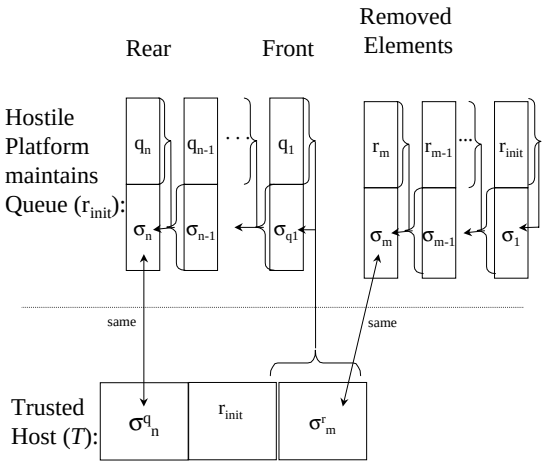

Figure 2: A resource-limited, secure implementation of queues.

$\mathcal{T}$	*computes and checks that*
	$\sigma_{front} = \sigma(x \mid\mid \sigma_r)$
$\mathcal{T}$	${\sigma'}_r \leftarrow \sigma_{front}$

$\mathcal{NB}$: *If* $\mathcal{H}$ *says "queue empty", ensure that* $\sigma_r = \sigma_q$

When the $\mathcal{H}$ gets a dequeue request, it returns the item ostensibly at the front of the queue, and a signature σ_{front}. $\mathcal{T}$ appends the returned item to its stored σ_r, signs the result, and compares the signature with σ_{front}. If the signatures match, it approves the operation, and updates σ_r to σ_{front}.

The queue protocol is illustrated in Figure 2. $\mathcal{T}$ retains two signatures: σ_q, which is an digest of the entire queue, and σ_r, which is a digest of all the items that have been removed. With each enqueue request, the $\mathcal{T}$ updates the σ_q to include the item in the queue digest. The $\mathcal{H}$ is asked to store the item and the current digest. We assume $\sigma(x)$ represents a keyed cryptographic hash or public key signature. Upon a dequeue request, $\mathcal{H}$ is asked to return the item at the front of the queue x, and the associated signed digest σ_{front}. $\mathcal{T}$ now uses the σ_r item value stored in trusted memory to authenticate the dequeued value: this signature represents all the items that have ever been removed from the queue. $\mathcal{T}$ compares a new ${\sigma'}_r$ value to the result of signing the string obtained by appending the item x claimed to be at the front of the queue to the old signature, σ_r. The following section examines the correctness of our protocol more closely.

4.1 Evaluation

We now argue that the queue protocol detects incorrect operation of queues by $\mathcal{H}$.

Definition 5 *An* ideal queue *is one which works according the usual LIFO discipline.*

As before, incorrect queues are ones which, (after some series of operations) on a dequeue, return an item other than the one which would be at the head of the ideal queue after the same set of operations.

Definition 6 *We define an* incorrect queue *as one which after some series of operations* $\Omega = o_1, \ldots o_n, dequeue(queue_{id})$ *returns a value different from the one that would be on the head of an ideal queue after the operations* $o_1, \ldots o_n$.

Definition 7 *A protocol for checking queue implementations is secure if an incorrect queue is always detected whenever it returns a wrong value on a dequeue.*

Here is the main claim of correctness of our queue protocol.

Theorem 2 *Our queue protocol is secure, as long as the signature scheme on which it is based is collision resistant and* 2^{nd} *pre-image resistant.*

We use the notion of a correct digest here as well; however, in the case of the queue, there are two pieces to the digest, σ_q, which represents the entire "historical" queue, including items that have ever been enqueued, and σ_r, which represents just all the items that have been dequeued (see figure 2).

Definition 8 *A correct digest of a queue with initializing value* q_0, *items* $q_1, \ldots q_n$ *that are currently on the queue (with* q_1 *being the item to be next dequeued) and items* $r_1 \ldots r_m$ *that have been removed (*r_1 *the first item removed,* r_m *the item most recently removed) consists of two signatures,* σ_q, σ_r *which are computed as follows:*

$$\sigma_{r_0} = \sigma(q_0)$$

$$\sigma_{r_i} = \sigma(r_i \mid\mid \sigma_{r_{i-1}}), \quad i = 1 \ldots m$$

$$\sigma_{q_0} = \sigma_{r_m}$$

$$\sigma_{q_i} = \sigma(q_i \mid\mid \sigma_{q_{i-1}}), \quad i = 1 \ldots n$$

Claim 3 *After any series of operations* $\Omega = o_1 \ldots o_n$ *Our protocol always maintains the correct digest of an ideal queue in* $\mathcal{T}$, *providing a)* $\mathcal{T}$ *operates correctly according to our protocol, and b) the underlying signature scheme is collision resistant and* 2^{nd} *pre-image resistant.*

We show this by induction: at initialization, the claim holds trivially. Now consider each queue operation:

Enqueue On an $enqueue(x)$ request, the σ_r is unchanged (by Protocol 2); this is specified by Definition 8. The σ_q is computed as follows (again, as specified by Definition 8):

$$\sigma'_q = \sigma(x \mid\mid \sigma_q)$$

Dequeue On an $dequeue(x)$ request, the σ_q is unchanged (by Protocol 2) as specified by Definition 8 above. The σ_r is updated as described in Protocol 2. The $\mathcal{H}$ returns a signature, σ'_r and an item x. The following equality is checked:

$$\sigma'_r = \sigma(x \mid\mid \sigma_r)$$

if the equality holds, then (assuming that the signature scheme that is used has the desired properties) setting σ_r to σ'_r will correctly update σ_r. Note that $\mathcal{H}$ does not compute σ'_r (since the signing key is secret and we assume collision resistant and 2^{nd} pre-image resistant signature schemes). This value is given to $\mathcal{H}$ at the time x is enqueued: if $\mathcal{H}$ returns that value, and it checks out, the correctness of the digest is preserved. Note that the way signatures are used here is different from the stack protocol. In the stack protocol, the $\mathcal{H}$ returns an item and an old signature (*i.e., inputs* to the signature algorithm) which when signed should yield a value identical to the digest held in $\mathcal{T}$. In the case of queues, the $\mathcal{H}$ should return an item and an *output* signature; the item, and the σ_r digest, when signed together, should match the output signature returned by $\mathcal{H}$.

The operations *delete* and *new*; will create a new, independent, (correctly initialized) digest of a new queue, and/or to terminate the current stack instance.

When the following claim is established, the theorem is proven.

Claim 4 *If the* $\mathcal{T}$ *always stores the correct digest of the ideal queue, after every sequence of operations* $\Omega = o_1 \ldots o_n$. *then an incorrect queue can always be detected, provided the underlying signature scheme is collision resistant and* 2^{nd} *pre-image resistant.*

Assume that after the operations Ω above, we execute a *dequeue*. Assume the $\mathcal{T}$ correctly digests the ideal queue after the operations Ω into σ_q and σ_r. Now

on the dequeue, $\mathcal{H}$ returns an item x and a new signature σ_r', which is verified as:

$$\sigma_r' = \sigma(x \mid\mid \sigma_r)$$

where x is the item currently on the head of the queue. Now, because of the collision resistance of signature scheme, and the cryptographic assumption that $\mathcal{H}$ is unable to compute the signature, $\mathcal{H}$ cannot feasibly substitute another x or σ_r'. Therefore, an incorrect queue operation will be detected via a bad signature.

5 Schemes for RAM and Trees

There have been several schemes proposed in the literature to handle a random access memory (RAM) [5]. Most of these schemes are based on Merkle signature trees [12]. We describe the signature tree and discuss the tradeoffs in implementing secure stacks and queues using them.

5.1 Prior Work on RAMs

Given an n-bit address space, one can construct secure RAM M as a binary tree with 2^n leaves and 2^n interior nodes, using 2^{n+1} data elements on an insecure memory array M^i (where "i" stands for insecure). Each bit of the address selects a branch on the binary tree. Figure 3 shows a (tiny) RAM with a 4 bit address space. Each node in the RAM is unambiguously designated by a bit substring of the address. The leaves store the values of the RAM. That is, given a (complete) n-bit address string a, and the buggy RAM, $M^i[a]$ is the actual value of the memory cell at address a. Tampering of these values by the adversary is deterred (with high probability) by storing signatures in the interior nodes. These signatures are computed as follows. For a given interior node with address a where $\mid a \mid < n$ (Note: we treat each bit string as a different index into the (insecure) memory array. Thus `0011` is not the same as `11`. This can be accomplished by a simple transformation of the bit string into an integer array index).

$$M^i[a] = \sigma(M^i[a \mid\mid 0] \mid\mid M^i[a \mid\mid 1])$$

The root value $M^i[0]$ is kept in the $\mathcal{T}$. When an address a is accessed, all n values on the interior nodes along the address path a, as well as the n additional values need to compute the signatures on the interior nodes, (as well as the root value) are also accessed. When an address is modified, all the signature values on the interior nodes along the address path are recomputed.

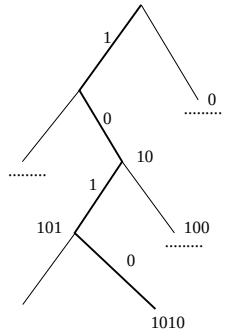

Figure 3: RAM: Address shown `1010`

5.2 Can we do just do Stacks & Queues on RAM?

Since RAMs are the most general type of memory, the question naturally arises, can we not simply implement secure stacks and queues using the secure RAM? The answer to this question in a given specific situations depends on a set of engineering design issues. The main consideration in using a RAM is the number of signature computations. Any time any particular (leaf) value is addressed in the binary tree representation, n signature computations are required. This could be avoided by partitioning the RAM into large *pages*, there by reducing the effective number of signature computations. Another approach is to retain a certain number of pages in the $\mathcal{T}$ and "swap in" pages from $\mathcal{H}$ only when needed. This would amount to a *secure virtual memory.*

There are several complications in implementing secure paging systems based on the RAM schemes described above. Limited program (text) space on a $\mathcal{T}$ device (*e.g.,* a smart card) may preclude implementation of a secure virtual memory. Limitations in the data memory will limit the amount of "trusted" pages that can be kept within $\mathcal{T}$. Bandwidth limitations will force high penalties for page faults. Decreasing the page size too much will increase the number of signature computations on each page fault. With an n bit address space, there will be $O(n)$ signature computations on a page fault: signatures will have to checked when an "outdated" dirty page is read, and recomputed upon write out. If another page is swapped in, another set of signatures will have to be checked.

If the application primarily uses stacks and queues, and the above mentioned complications dominate, then our stack and queue schemes will be useful. However, if a high-quality implementation of a virtual secure memory is available, then it would be reasonable to use that. However, the authors are not aware of

any such implementations for $\mathcal{T}$ devices currently on the market. Unlike secure virtual memory schemes, which must be carefully implemented and tuned, our schemes are relatively simple, and can be built by an application programmer.

6 Applications

The problem of checking large data structures with limited memory was motivated by new applications for software tools [7, 6]. The goal of this work is to place trusted software tools (static analyzers, type checkers, proof checkers, compilers/instrumenters, etc.) in trusted hardware; the output of these tools would be attested by a signature in a public-key crypto system.

One particular application concerns Java^{TM} bytecode verification [11]. This is a process similar to typechecking that is carried out on Java^{TM} virtual machine (JVM) programs. The JVM is a stack-oriented machine. The typechecking process ensures that for every control flow path leading to a given point in a JVM program, the types of the stack entries are compatible: *e.g.* an object of type `NetSocket` is never conflated with an object of type `Circle`. There are other properties that bytecode verification ensures, but the key *typesafety* property is at the core of the security policies of the JVM. Currently, this process is carried out by browsers such as Netscape^{TM} prior the execution of mobile Java^{TM} code such as applets. Performance, security, configuration management, and intellectual property protection advantages are claimed when bytecode verification and similar static analysis processes are conducted by a a trusted $\mathcal{T}$ machine. The result of the analysis is attested by a (pubic-key based) cryptographic signature on the mobile code.

The bytecode verification algorithm [11] involves, *inter alia* a) maintaining an agenda of control flow paths to be expanded, b) computing the state of evaluation stack, c) looking up the typing rules for various types of instructions d) maintaining symbol tables of variables. This resource-intensive data usage taxes the resources of all but the most powerful (and expensive) $\mathcal{T}$ devices. Another application discussed in [6] is placing a proof checker in a $\mathcal{T}$ device. Necula [13] suggests that mobile code should carry with it a proof of an applicable safety property. Unfortunately, such proofs reveal a great deal about data structure and layouts, loop invariants and algorithms. Vendors may balk at revealing such intimate details of their products. With a proof checker in a $\mathcal{T}$ device, the vendor can check the proof at their site. The bare binary (sans proof) can be signed by the $\mathcal{T}$ device to attest to the correctness of the proof, which can then remain secret. Yet another application is the creation of trusted, signed analysis products (control dependency graphs, data dependency graphs, slices etc.) to accompany mobile code. Such trusted analysis products can be used by security environments to optimally "sandbox" [15] mobile code.

Techniques such as the ones we suggest will form a useful implementation technique in placing trusted software analysis tools in trusted hardware. While trusted hardware devices can be expected to become more powerful, the inherent physical design constraints (form factor, energy usage, heat dissipation) are likely to prevent the performance gap (with conventional machines) from narrowing; so implementation techniques such as ours will remain applicable.

7 Extensions and Future Work

The techniques we have discussed above simply insure the integrity of stacks and queues. We now discuss some extensions and future work to address some related issues such as confidentiality, key expiration, and data structure sharing.

Providing Confidentiality Confidentiality may be a concern in applications of stacks and queues. Our protocol designs allow for layering (or integrating) confidentiality mechanism on top of the basic protocols.

Updating Keys Since we assume a computationally bounded adversary, cryptographic keys used to protect the integrity of the data structure have a limited lifetime. Relying on keys beyond their lifetime could compromise the integrity of an instance of the data structure. We now discuss how to manage keys for instances of data structures.

One general approach of replacing keys is the most obvious: *complete rewrite*. That is to temporarily suspend normal usage of the data structure to remove all elements from one instance of the data structure and add them to a new instance with a new key. Some issues arise from this approach. It may not be possible to go directly from one stack to another since the order of the elements would be reverse. However, this problem can be corrected by repeating the process with another stack. Alternatively the operation can be performed in a single pass using a queue.

A second general approach for updating keys is *gradual transition* from one key to another. This can eliminate the need for the data structure to be unavailable during key updates. That is, multiple signatures using different keys are maintained until the entire structure has been updated with the new key.

Sharing Data Structures Multiple entities may wish to share operations to a data structure. Issues involved in sharing the data structure concern sharing the most recent signatures and keys associated with the structure. Such schemes may be built on top of secure quorum schemes. However, it is unclear whether such schemes satisfy the security and performance requirements for sharing data structures. This area is a topic of future research.

8 Conclusion

We have described protocols by which resource-limited, trusted computers can store stacks and queues on untrusted hosts while retaining only a constant amount of memory in the trusted machine. Tahis approach differs from earlier work [4, 5]. Our protocols are much simpler. We use $O(1)$ memory in the trusted computer, and transfer only $O(1)$ amount of data for each push and pop operation in an on-line mode. Previous approaches used a $O(log(n))$ trusted memory and $O(log(n))$ data transfer for each operation (where n is the size of the stack or queue). However, unlike previous approaches, which use information theoretic bounds, we assume computationally limited adversaries. We present arguments to show that our protocols will detect attacks which return incorrect values.

9 Acknowledgements

This work has greatly benefited as a result of early discussions with Dave McAllester on RAM trees, more recent discussions with Philip Fong on associative arrays, as well as feedback from the anonymous reviewers of this conference.

References

[1] *Javacard 2.0 Application Programming Interfaces, Sun Micro-systems, Inc., October 13, 1997.* (See also: `http://java.sun.com/java/-products/javacard`).

[2] *Spyrus Product Guide, Spyrus, Inc.* (See also: `http://www.spyrus.com`).

[3] *The Mondex Magazine, Mondex International Limited, July 1997.* (See also: `http://www-.mondex.com`).

[4] Nancy M. Amato and Michael C. Loui. Checking linked data structures. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1994.

[5] Manuel Blum, William Evans, Peter Gemmell, Sampath Kannan, and Moni Noar. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994. Originally appeared in *FOCS* 91.

[6] P. Devanbu, P. W. Fong, and S. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the Twentieth International Conference on Software Engineering*, 1998.

[7] P. Devanbu and S. G. Stubblebine. Cryptographic verification of test coverage claims. In *Proceedings of The Fifth ACM/SIGSOFT Symposium on the foundations of software engineering*, Zurich, Switzerland, September 1997.

[8] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual Symposium on Theory of Computing*. ACM, 1987.

[9] John V. Guttag and James J. Horning. *LARCH: Languages and Tools for Formal Specification.* Springer Verlag.

[10] L. Lamport. Password Identification with Insecure Communications. *Communications of the ACM*, Nov 1981.

[11] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine specification.* Addison Wesley, Reading, Mass., USA, 1996.

[12] R. C. Merkle. A certified digital signature. In *Advances in Cryptology–Crypto '89*, 1989.

[13] George Necula. Proof-carrying code. In *Proceedings of POPL 97.* ACM SIGPLAN, 1997.

[14] R. Ostrovsky. Efficient computations on oblivious rams. In *Proceedings of the 19th Annual Symposium on Theory of Computing.* ACM, 1990.

[15] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, 1993.

[16] Bennet Yee and Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.