Department of Computer Science
University of Colorado
Boulder, CO 80309-0430

A Tamper-Resistant Programming Language

Dennis Heimbigner
dennis@cs.colorado.edu

Technical Report 931-02

May 20, 2002

Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430

A Tamper-Resistant Programming Language

Extended Abstract

1 Computing in a Hostile Environment

An important and recurring security scenario involves the need to carry out trusted computations in the context of untrustworthy environments. One approach is to combine a secure co-processor [11] with an untrusted host computer. The secure processor provides the environment in which to perform trusted computations, and the insecure host provides additional resources that may be used by the trusted processor. Unfortunately, there is no guarantee that the host will not tamper with the resources used by the secure processor in an attempt to corrupt the operation of the secure processor.

This paper demonstrates how a programming language – specifically Lisp 1.5 – can be used to provide a convenient and general mechanism for tamper-resistant utilization of a specific resource, namely the memory of an untrusted host. This means that any program written in the language and any computation carried out by such a program is inherently resistant to tampering, and if tampering occurs it will be detected *on-line* (before the computation is complete) so the computation can be aborted.

This language-based approach is in contrast to prior work, which has focused on providing specific tamper-resistant data structures. These structures and implementing code are stored in the host's memory and have the property that any attempt by the host to tamper with the data structure will be detected. Examples of such data structures include random-access memory [3], simple linear lists [1][3], and stacks and queues [3][4].

The language approach has several advantages compared to the data structure approach. It is more general since the programmer can use any data structure that can be implemented in the language. Additionally, it hides the complexity of tamper resistance. Programmers do not have to worry about the problem of tampering because a solution is built into the language implementation and is inherited by all programs executed by the language interpreter. As a result, it should be easier to construct programs that can avail themselves of untrustworthy host memory.

This solution also differs from a programming system such as the Java SmartCard environment [9]. The solution proposed here provides a cross-platform programming system in which an interpreter for the language resides on the secure processor but the programs and data reside in the untrusted host. Java SmartCard provides a version of Java in which code and data reside entirely on the SmartCard.

This language-based approach is practical when the secure processor has adequate computational power as well as bandwidth to the host memory. A secure processor such as the IBM 4758, for example, can access the host memory at near bus speeds, and could provide a practical platform for a tamper-resistant Lisp. The currently available Java SmartCard, on the other hand, is slow and has limited bandwidth, and probably could not provide a practical platform.

In order to limit the scope of the problem, only the issue of integrity is addressed in this paper; the issue of confidentiality is deferred. It seems reasonable, however, to assume that adding confidentiality is a straightforward application of encryption to the values stored in the host memory.

2 Why Lisp 1.5?

Lisp 1.5 was chosen as the target programming language primarily because of its simplicity. It provides a usable, if not particularly efficient language. It has a simple and small interpreter that can easily be implemented on a secure co-processor with limited resources. Equally important, Lisp uses lists as its only data structure, and it is relatively easy to implement tamper-resistant lists. Additionally, both programs and data are represented as lists; hence tamper resistance can be applied to both code and data with no extra effort.

This paper assumes general familiarity with the Lisp 1.5 language and its implementation. A complete review of Lisp 1.5 and its original implementation is available from the "Lisp 1.5 Programmer's Manual" [5] by John McCarthy et al. The most important issues with respect to Lisp (and its implementation) are the following.

- Data Structures – How do we represent list cells, atoms, and primitive values (e.g. integers)?
- Tamper Checking – When and how is tamper checking performed?
- Garbage Collection – How do we implement garbage collection over the memory of our untrusted host in such a way as to guarantee tamper-resistance?

3 Lisp List Representation

In Lisp, the basic unit of memory is a "cell", which in this variant of lisp contains the following fields.

- Car, Cdr – two pointers to other cells. Lists are constructed by linking cells using these pointers. Traditionally, the "list" is considered to be the set of cells reached by following the Cdr pointers. Cells reached through the Car pointer are often referred to as "sublists".
- Flags – a number of bits are set aside in the cell to mark special states. In particular, there will be an *atom* flag to indicate that that this cell conforms to the format of an atomic (i.e., non-list cell) value. Additionally, there will be several flags for use by the garbage collector (Section 6).
- *s*, *m* – two fields are added per cell to contain hash values for signing the cell contents to achieve tamper-resistance. These are not standard Lisp fields.

Figure 1 shows a set of linked cells with one list beginning with the cell labeled C1. The list beginning with cell C4 is a sublist of two lists. Cell C5 illustrates the simplest possible example of a cyclic list in which the Car field of a cell points to the cell itself. As is usual with Lisp lists, lists (and sublists) are terminated by the *Nil* value (indicated by a diagonal line) in the Cdr of the last cell. In addition to list cells, a simple, traditional Lisp also supports *Atoms*, which are considered primitive non-list values. Nil and Integers are also considered to be Atoms. Integers are represented by storing the integer value directly in a cell. A flag indicates that the cell is an integer and that the Cdr field contains the integer value. This makes the reasonable assumption that an integer and a pointer have the same size. Figure 1 shows two integers, 79 and 101. For the latter the dotted line points to the integer representation. Named atoms, such as "A" and "B", are also represented by cells with special content and flags. Figure 1 shows "A" and "B", and for the latter, a dotted line points to its representation, which consists of a specially flagged cell and two pointers. The Car field points to the print name of the atom and is represented as a list of integers where each integer is the value of a character of the name. The Cdr field of an atom cell is used to create a list of all known atoms. New atoms are added onto the front of this list. We assume that atoms can be created but never destroyed. The root of this list is kept in the memory of the secure processor.

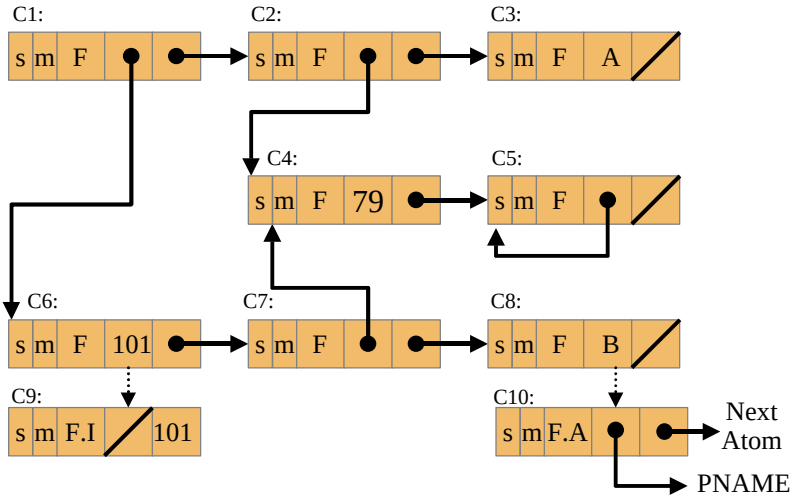

Figure 1. Example List Structure

4 Secure Processor – Host Access Protocol

The secure processor accesses the host through the following primitive operations.

1. *read(i,c)* – requests the host to return the cell located at host memory address *i*. The data is returned in *c*.

2. *write(i,c)* – requests the host to store the cell contained in *c* at the host memory address *i*.

3. *alloc(n,i)* – requests the host to allocate *n* sequentially located cells of new memory and return the address of the start of that memory in *i*.

4. *release()* – requests the host to reclaim all memory that was allocated to the secure co-processor. It is assumed that these operations are implemented in terms of some underlying protocol that supports communication between the secure processor and the host.

This interface also defines the attack model. It is assumed that any values kept in the memory of the secure processor cannot be read or modified by the untrusted host. The only mechanism by which the host can tamper with a computation of the secure processor is through the values the host returns in response to read requests from the secure processor.

5 Basic Approach to Tamper-Resistance

Tamper resistance is achieved by dividing the whole computation (the program execution) into *epochs*. Each epoch has an associated index that acts as a timestamp for all write operations performed during that epoch. Epoch boundaries are defined by occurrences of garbage collection. Thus, every time the garbage collector is invoked, a new epoch begins. The sequence of epochs continues until the computation is complete.

Within a given epoch, the following operations encapsulate communication between the secure processor and the host.

- *CAR(p), CDR(p)* – these operations take a pointer *p* to a cell, read the cell (with tamper checking) and extract either the Car or Cdr fields respectively.

- *CONS(p,q,f)* – this operation constructs the contents of a cell with *p* in its Car field, with *q* in its Cdr field, and with *f* in its flag field. It then obtains a pointer to a free cell and writes the newly constructed cell contents into that free cell.

- *new()* – this procedure returns a pointer to an unused cell. For now, we will assume that such a *free* cell comes from the *freelist*, which is a special list of unused cells linked together through their Cdr fields and with a special *free* flag set. They are checked for tampering when they are removed from the freelist. If all space is exhausted, then an *alloc()* request is made to the host to obtain more memory.

Tampering with a cell's contents is detected by adding a cryptographic signature as a new field in each cell (the *s* field in Figure 1). The signature is computed using any reasonable cryptographic one-way hash function such as SHA-1 [6] or MD5 [6]. The hash function takes the *ordered* concatenation of the following four values as its input.

1. Cell contents – the Car, Cdr, and Flags fields (also ordered).

2. Cell address – the address from which the cell was read.

3. Time stamp – the current epoch index.

4. Secret key – a key known only to the secure processor.

Whenever a cell is read, the signature is recomputed and if it matches the stored signature, then it is assumed that the Car, Cdr, and Flag fields are valid. The *s* signature field allows the contents of a cell to be validated using only the address of the cell, the contents of the cell, and the key and epoch index information contained in the secure processor.

It is important to note that in the absence of encryption, the security parameter for this signature is not determined by the total input size (512 bits), but rather by the size of the secret key (currently 96 bits). The key size can be increased to 128 bits by using the technique in Section 12.2. As a consequence, brute force attacks on the signature are possible, but are assumed to be hard. This is in line with prior work [4], which assumes the adversary has limited computational power. Information theoretic bounds [1][3] are not considered here.

This signature serves two important functions. First, it prevents an attacker from synthesizing arbitrary cell contents. This is because the key is kept secret and cannot be discovered by inverting the hash function or accessing the memory of the secure processor. Second, a valid signature guarantees that the interpreter wrote the contents to that cell at some point in time. Note that this does not by itself prevent a replay attack since any value written to that cell will have a valid signature.

The important point to note is that during an epoch, any given cell in the host memory will be written at most once. We will refer to this property as *write-once-per-epoch*. This property is enforced by the fact that the only memory writing that can occur is through the *CONS* procedure, and it is defined to always store its result in a new cell taken from the freelist.

Within an epoch, replay attacks cannot occur because of the write-once-per-epoch policy. This means that an attacker can return only that single – correct – value written in the epoch. Since the signature includes the cell address and the epoch index, it is not possible to attempt a replay involving either the contents of another cell or the contents of the same cell from another epoch.

6 Garbage Collection

The transition from one epoch to the next is a side effect of garbage collection. Garbage collection is expected to have two specific effects upon its completion.

- All unreachable cells have been linked into a single free list.
- All reachable cell signatures have been updated based on the new epoch index.

There are two traditional approaches to garbage collection: reference counting and mark-and-sweep. Reference counting as a means of finding unreachable cells was considered and rejected because it requires the reference count to be modified many times. These multiple writes could provide many opportunities for replay attacks in which the host provides an earlier version of the contents of a cell that contains an incorrect reference count.

Instead, a mark-and-sweep algorithm was chosen for garbage collection because it isolates the garbage collection activity into a single phase for which special anti-tampering mechanisms can be used. The mark phase involves walking all cells reachable from some identified set of root cells. The sweep phase then walks all cells in address sequential order, and when an unmarked cell is encountered it is added to the free list.

The mark phase operates by doing a depth-first traversal of the graph of cells reachable from a defined set of *root* pointers kept in the secure processor. One such root has already been mentioned, namely the root of the named atom list. As each cell is first reached, it is marked. At the point where we last touch a cell in the walk, we re-sign the contents using the new epoch index. Thus at the end of the traversal we have touched and re-written all the cells that are still reachable and in use. Since they have been re-signed using the new epoch index, they have effectively been moved into the new epoch. A given path ends when it encounters an Atom or encounters a cell that has already been marked. This latter case can occur either because some

cells may be reachable by more than one path during the walk or because the list is cyclic. Cells C4 and C5 show these two cases respectively.

The specific marking algorithm used here is the Schorr-Waite(-Deutsch) [8] algorithm for marking because it avoids the need for a separate stack. As it performs its depth-first walk, this algorithm temporarily reverses the list structure of the lists on the current path of the walk. During the marking process, a cell can be in one of four states.

1. *Unmarked* – any cell not yet reached during marking will be in the just completed epoch and no garbage collector related flags will have been set in the cell.

2. *Cdr Chaining* – some cells on the current depth-first path will have their Cdr field reversed and will have a flag set to indicate that fact. In addition, such a cell will have its mark flag set. It is still considered to be in the just completed epoch.

3. *Car Chaining* – some cells on the current depth-first path will have their Car field reversed and will have a flag set to indicate that fact. In addition, such a cell will have its mark flag set. It is still considered to be in the just completed epoch.

4. *Complete* – any cell for which Car and Cdr chaining is completed will be in the new epoch and will have its "marked" flag set.

Figure 2 shows a point in the traversal of the lists shown in Figure 1. The dotted lines indicate the boundary between the secure processor and the host. The box labeled P1 at the left represents a root pointer in the secure processor. The depth first walk starting at P1 has visited the top row of cells (C1, C2, C3) using Cdr reversal, and the subgraph rooted at cell C3 has been completed. Cell C2 is now doing Car reversal and cell C4 is starting the next Cdr reversal chain. Thus the current depth-first path is C1, C2, C4. The boxes labeled M1 and M2 represent special pointers in the secure processor. They are used to track the state of the marking procedure. Thus, M1 points to the last cell in the depth-first path (C4) and M2 points to the next cell to be marked (C5).

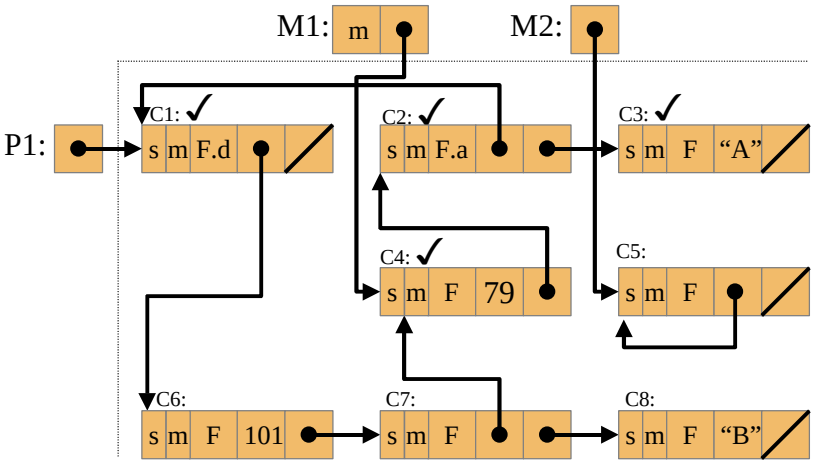

Figure 2. Schorr – Waite Marking

After marking is completed, the sweep phase examines every cell in the sequential order defined by its memory address. If the cell is unmarked, then it is garbage and it is marked as a free cell and is linked to the freelist. In order to avoid a second sweep to reset the mark bits, the secure processor just inverts the sense of the mark bit so that in the next garbage collection, all cells will be considered unmarked.

7 Adding Tamper Resistance to Garbage Collection

The garbage collection phase violates our write-once-per-epoch assumption and so it offers significant opportunities for tampering. Replay attacks are especially tempting because each cell will have been written several times.

The mark phase can be made tamper-resistant if we can guarantee that an untrusted host cannot successfully modify or replay cell values during the marking phase. This can be achieved by treating the current depth-first path with its reversed pointers as a stack. This stack can be made tamper-resistant by keeping a Merkle hash [7] of the stack of values, which is essentially the approach used in tamper-resistant stack data structures [4]. The *m* field of each cell is used to store this hash. Each cell in the depth-first path is modified to store the Merkle signature of its predecessor in the path. This signature includes all of the contents of the predecessor cell: the Cdr, Car, Flags, *s* and *m* fields. The root of this hash is kept as part of the M1 register in the secure processor's memory and is assumed to be unmodifiable by the host. In effect, each Merkle hash is actually a hash of all of the contents of all the cells below it in the stack. Any attempt by the host to modify or replay one of these cells will be detected upon return to that cell during the depth-first walk.

During garbage collection, the *s* signature field is recomputed every time a cell is modified. It remains a signature of only the cell address, the Car, Cdr, and Flag fields, and of the key and the index of the just completed epoch. When traversal of a cell is completed, the last action is to clear its Merkle signature *(m)* field and to recompute its *s* field to contain its signature but using the new epoch index.

During the sweep phase, each cell is read in turn and based on its contents, is either ignored or linked to the freelist. Verification of the contents of the cell depends on the contents of the cell.

- If the cell appears to be unmarked, then its signature is validated using the old epoch index. If valid, then the cell is rewritten with a flag indicating that it is free. Its Cdr field is used to link it to the front of the freelist. The hash field of the free cell is computed on the revised contents and using the new epoch index.

- If the cell appears to be marked, then its signature is validated using the new epoch index, and otherwise ignored.

The claim is that at the end of garbage collection, every cell is either on the free list or has been marked. In both cases, the cell has been re-signed using the new epoch index. At this point, the next epoch starts and computation resumes.

8 Replay Attacks Against Garbage Collection

While a keyed, strong, one-way hash function prevents the host from synthesizing arbitrary new cell contents, it does not necessarily prevent replay attacks. The use of the Merkle hash does prevent replay against the current path in the depth-first walk. But there is a potential problem. As Figure 1 shows, it may be possible to reach a cell, such as C4 or C5 by more than one path. Under normal circumstances, this would cause the walk to encounter an already marked cell, which in turn would cause the walk to stop and back up to continue the walk down another path.

However, whenever a cell is read from the host memory, a malicious host has the option of providing a replay of any of the four values stored in that cell during garbage collection.

1. It can replay the correct contents of the cell.

2. It can replay the contents of the cell as it was when involved in Cdr chaining.

3. It can replay the contents of the cell as it was when involved in Car chaining.

4. It can replay the contents of the cell as it was before garbage collection began.

Obviously Case 1 causes no problem. Cases 2 and 3 are a problem during the mark phase. Fortunately, if the attacker replays either of these values, then the mark flag will be set and will cause the garbage collector to properly stop its marking and back up to a new depth-first path.

Case 4 is a problem both during the mark phase and the sweep phase. During marking, the legitimate contents of the cell may indicate the cell has already been marked. If the host instead replays the unmarked cell contents, then the garbage collector will happily re-mark that cell and everything reachable from it as long as the host replays unmarked cell values. If a cyclic node is re-marked, then of course this will be detected by a failure to validate the *m* field if the current path backs up to that node. Otherwise, this re-marking by itself causes no harm because marking is an idempotent operation.

During the sweep, the same thing may happen. That is, the host may replay the unmarked contents of each cell. In this case, an active cell will be treated as a free cell and added to the freelist.

9 The Solution: So What?

The only possible effects of replay are to cause the mark phase to re-mark cells or to cause the sweep phase to free cells that are really reachable. The effects of both attacks are controllable.

- If the attacker always replays unmarked cell values during marking, then there is a potential for the mark phase to loop forever in the presence of any cyclic lists. Fortunately, the mark process knows that the maximum number of cells to mark is bounded by the number of cells allocated by the host using the *alloc()* operation. If the mark process ever discovers that it has marked more cells than have been allocated, then it can terminate and indicate that tampering has occurred.

- If the sweep phase erroneously frees reachable cells, then by definition that cell is pointed to by either another cell or by a root in the secure processor. Since such pointers can never legitimately point to a free cell, any attempt to read the erroneous cell will be detected and will cause the computation to abort.

Thus the effects of replay can be detected in time to prevent uncontrolled looping of the garbage collection process and to prevent corruption of the subsequent computation.

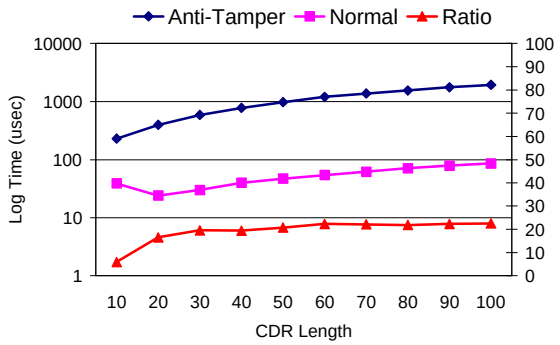

Chart 1. CDR Traversal

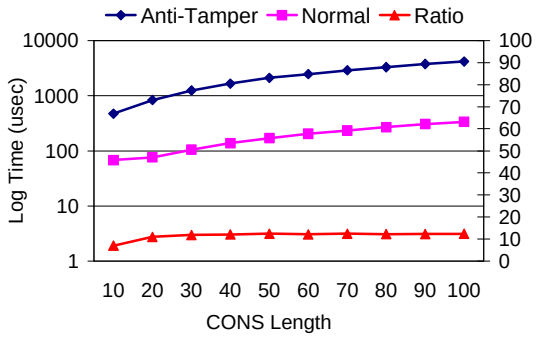

Chart 2. CONS Construction

10 Preliminary Performance Measurements

A complete implementation of the tamper-resistant Lisp system was not yet available at the time this paper was written. Key components of that system have been prototyped, however. These components include the interface to the Host, the *CAR*, *CDR*, *CONS* interface, and the mark phase of the garbage collector. Charts 1, 2, and 3 show some preliminary performance measurements produced by these prototype components. In lieu of a secure processor, the programs were executed using g++ version 2.95.3 on an Ultra-2 Sparc. The MD5 signing code is completely in software and uses a public domain version from RSA.

Chart 1 shows an average time (in microseconds of clock time) for traversing the Cdr link of lists ranging in length from 10 to 100 cells. Note that the left hand Y-axis is logarithmic (as it is for the other charts). The top line represents the time to traverse the list when the anti-tampering code is in place. The next line down shows the same traversals without the anti-tampering code. The bottom line shows the ratio of the other two lines: about 22 in this case (plotted using the linear right hand Y-axis). Although not shown, each traversal requires one invocation of MD5, a signing, to verify each cell as it is read for traversal.

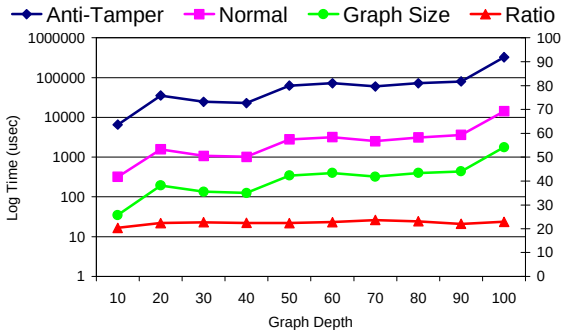

Chart3. Graph Marking

Similarly, Chart 2 shows an average time for repeatedly applying *CONS* to construct lists ranging in length from 10 to 100 cells. Again, the top line is the anti-tamper version, the second line is the normal version, and the third line is the ratio: about 12. Each *CONS* operation requires two signings: one to verify the free list when obtaining a free cell, and the other to sign the new cell contents when it is written to the (simulated) host memory. The ratio is smaller than in Chart 1 is because the normal *CONS* takes substantially more time to compute than the normal *CDR* code, and a small change in this already small value causes a large change in the ratio.

Chart 3 shows an average time for doing the garbage collection mark phase on acyclic graphs whose depth ranges from 10 to 100. These graphs are generated with random widths in each level (up to a maximum of 64 cells) and the number of roots is chosen randomly in the range 1 to

8. The top two lines are the anti-tamper and normal times as before. The bottom line is the ratio (about 23), and the next to the bottom line is the total number of cells in the graph.

It should be clear that these measurements are preliminary, and several factors must be considered when evaluating them.

- The absolute performance numbers are of only limited interest since they depend heavily on the processor speed and the speed of signing.

- The high ratios (12-20) indicate that the signing times dominate all other factors. Thus, any mechanism that can reduce the signing time will have a major impact on the relative performance of the anti-tamper version vis-à-vis a normal implementation. To take an extreme example, if very fast signing hardware is available and can speed up signing by a factor of 10, then the ratio between the two implementations becomes quite acceptable. Reducing the hash input block size and the output digest size can also reduce the signing cost. This increases the vulnerability to brute force attacks, but if the computation has a very limited lifetime, then this may be acceptable.

- The memory bandwidth between the secure processor and the host has a significant impact on the overall performance. Not only is every cell read by the secure processor, the cell size increases significantly. The addition of the two signature fields, each 128 bits long, causes a 300% increase in the memory size of a cell.

At some point, it will be necessary to move the implementation onto actual hardware or onto an accurate simulator for such hardware in order to achieve more realistic performance measurements.

11 Miscellaneous Issues

The assumptions underlying tamper-resistance affect a number of less important issues in implementing the Lisp interpreter. These include direct cell modification, Lisp binding mechanisms, the use of a non-recursive interpreter, and memory allocation management.

11.1 REPLACA/REPLACD

Lisp often defines two special operations that can directly modify the contents of the Car or Cdr fields of a cell. These operators are *REPLACA* and *REPLACD*. Unlike *CONS*, these operators do not allocate their result in a new cell, but rather modify an existing cell. These operators are not implemented because they directly violate write-once-per-epoch.

11.2 Bindings

The evaluation of a *Lambda* expression causes the list of parameter variables (atomic names) to be bound to the associated argument values. As with traditional Lisp implementations, this version uses the ALIST, which is a list of pairs of the form (parameter,value). Given the ALIST, the current value of an atom is found by looking up the atom in the ALIST and using the associated value. This is the traditional shallow-binding model. When the evaluation of a Lambda expression is complete, the bindings associated with that expression are “popped” from the ALIST revealing any previous bindings. This requires a “frame stack” to mark the sequence

of bindings on the ALIST. In this implementation, a separate list is kept that contains the frame markers into the ALIST. Root pointers are kept in the secure processor for both the ALIST and the frame stack.

In addition to the ALIST, atoms can have a global binding called the APVAL. The APVAL for an atom can be set directly using the SETQ function. Usually, the APVAL is set using REPLACA to directly modify a cell associated with the atom, but this approach cannot be used in this implementation because it violates write-once-per-epoch. The solution is to keep another rooted list similar to the ALIST but which binds atoms with their APVAL. Retrieving the APVAL of an atom involves looking up its name on the APVAL list. Changing the APVAL for an ATOM just prepends the (atom,value) pair onto the APVAL list. This approach is less efficient than direct modification of the APVAL location. It also increases the space requirement because the wasted cells on the APVAL list are still reachable and hence will not be garbage collected. Periodic reconstruction of this list can solve the space problem and decrease the search times.

11.3 Non-Recursive Interpreter

A naïve implementation of the interpreter in the McCarthy book requires a recursive interpreter on the secure processor. Since this processor is assumed to have bounded memory resources, it must be implemented without recursion in the secure processor. Recursion can be avoided by rewriting the interpreter to always do tail recursion. This can then be implemented using branch instructions. The recursion involved in executing Lambda expression is handled by the use of the frame stack and ALIST mechanism described in Section 11.2. Any other essential recursion can be managed by creating and storing stacks in the Host memory.

11.4 Memory Allocation Management

The interpreter must periodically invoke the *alloc()* operation (Section 4) in order to obtain additional blocks of memory from the host. There is no reason to assume that these blocks will be contiguous, and so the interpreter must keep track of them in order to perform the sweep phase of garbage collection. We assume that a rooted SBREAK list is maintained as a list of sublists, where each sublist stores the size and address (as integers) of an allocated block. Note that this list can be kept with the rest of the Lisp cell memory since its interpretation as integers avoids any circularity.

12 Optimizations

The Lisp programming system described in this paper represents a proof of concept. In order to simplify the presentation, a number of identified optimizations are not included. This section provides brief descriptions of some of these optimizations and their merits and demerits.

12.1 Acyclic Graphs

The *CONS* operator cannot produce cyclic lists since it stores its arguments in a new cell. One way to introduce cycles is through *REPLACA* and *REPLACD*, but these are not implemented in this system. The only other way is through some direct action by the interpreter, but it turns out that this is unnecessary for implementing Lisp 1.5. Therefore it is possible to assume that no list cycles exist. Under that assumption, it is fairly easy to show that the signature field and the

Merkle signature field of a cell can be combined. During the depth first walk in the mark phase of garbage collection, a single signature field is used to hold the Merkle signature. At all other times, the field holds the normal signature. This can reduce computation time because the number of signatures computed is reduced. It also reduces the memory overhead by removing one of the 128 bit signature fields from the cell.

12.2 Combining the Epoch and the Key

The epoch indices need not be sequential since only two are ever used at any point in time, and then only during garbage collection. So, the epoch index can be any non-repeating sequence of random numbers. This suggests that the epoch index can also be used as the Key in the hash function. A small amount of space is saved, and the signing time does not change. The two primary advantages to this optimization are that the key size can be increased to 128 bits, and it provides a natural mechanism for re-keying by producing a new secret key for every epoch. This can reduce the possibility of brute force attacks against the hash function by the untrusted host.

12.3 Generational Garbage Collection

Any mechanism that can reduce the number of cells read during the mark and sweep phases will improve performance. One possibility suggested to the author is to use a generational garbage collector. After some number of epochs, some cells are permanently marked and hence need not be traversed during the mark phase. If compacting can somehow be included, then the sweep phase can also avoid these cells.

12.4 Host-based Garbage Collection

It has also been suggested that it may be possible to completely off-load garbage collection to the host. The primary advantage is that it significantly reduces the size of the interpreter and hence reduces the amount of code that must be in the trusted computing base on the secure processor. This alternative does have a number of disadvantages.

- Encryption is impossible because the host must know the contents of cells in order to perform the garbage collection function.

- The epoch mechanism cannot be used because the secure processor needs to determine which cells to re-sign and this requires walking the cells, which defeats the purpose of off-loading the collector.

- The secure processor contains several root pointers whose content is essential to garbage collection. Thus, the host must have some way of synchronizing with the secure processor to get the latest contents of the root pointers.

- The host interface must include some form of *newcell()* operation that returns monotonically increasing cell addresses. This imposes a rather large burden on the host since it must maintain a map from the addresses known by the secure processor to the actual addresses of those cells in the host memory. Repeated garbage collections, possible compacting, and the non-locality of Lisp lists all could conspire to make this map large: possibly with an entry for every cell.

In spite of these limitations, this optimization is intriguing, and it would be worthwhile implementing and measuring the effect of this approach to garbage collection.

13 Related Work

The approach proposed in this paper is directly inspired by the prior work in tamper-resistant data structures. Their focus was on defining data structures with the following properties:

- The data is stored in the untrusted memory of host computer,
- The code implementing the data structure is also in the host,
- The interface between the secure processor and the host is defined by the operations supported by the data structure. A stack, for example provides an interface consisting of *pop(), push(), top(), isempty(), new(), and delete()* [4]

The algorithms and data structures are designed to allow the eventual detection of tampering. The term "eventual" generally has two interpretations. The detection may be *off-line*, which means that verifying the absence of tampering occurred after the computation was complete, but before, presumably, the result is used in any consequential action. Alternatively, detection may be *on-line*, which means that verification occurs in parallel with the computation so that tampering can be detected before the computation is complete. In either case, there is some associated computational cost in space and time.

As indicated in Section 1, the language approach appears to be more general and convenient than the data structure approach. The set of data structures is not limited, and there is no need to modify the structure of a program in order to take advantage of a specific tamper-resistant data structure.

It may be claimed that using Lisp as the programming language is also using a single data structure, namely lists, but providing the additional elements of a complete programming system (interpreter, garbage collector, and storage allocator), raises the level of abstraction provided to the programmer by providing a comprehensive solution.

This work explicitly assumes the use of secure hardware whose memory cannot be read or modified by the untrusted host. Software-only solutions to the trusted computing problem exist [10] that use various forms of obfuscation to prevent an untrusted software program from analyzing the actions of the trusted software. Recent theoretical results cast doubt on the generality of this approach, and indicate that completely general software-only solutions may be impossible [2]. Nevertheless, applying software obfuscation techniques to the problem of a tamper-resistant programming language may be a fruitful avenue of research.

14 Summary

This paper proposes the use of a tamper-resistant programming language because of its many advantages over other approaches. The claim that the implementation is tamper-resistant rests on the following five arguments.

1. An attacker (the host) can never undetectably provide synthetic data to the secure processor. This is because all data is signed using a keyed cryptographically strong signing function. The best that the host can do is replay data written to its memory by the secure processor at some time in the past.

2. Write-one-per-epoch combined with signing of cells guarantees that an attacker cannot successfully replay data during an epoch.

3. The cells in any path constructed by the depth first walk during marking cannot be modified or replayed except with the original unmarked contents because of the Merkle hash maintained over the current path.

4. The host can cause the garbage collector mark phase to re-mark cells by replaying unmarked cell data, but the cost of this is bounded because there is checkable upper-bound on the total amount of re-marking required by garbage collection.

5. The host can cause the sweep phase of garbage collection to mark reachable cells as free, but this will be detected the first time that such a cell is read by the secure-processor.

A complete implementation of the tamper-resistant Lisp is in progress. Future research will examine possible applications of this technique to RAM programming models. Moving from Lisp to Java, for example, requires solving the problem of efficient tamper-resistant access to linear random access memory in the form of arrays. Traditional approaches use Merkle hash trees that effectively double the memory requirement and convert the access speed from *O(1)* to *O(log n)*. There is no reason to believe that this burden is escapable, and so it will be necessary to find an alternative approach that amortizes the cost in a different way using some variant of the epoch mechanism.

References

[1] Amato, N.M. and M.C. Loui, "Checking Linked Data Structures," Proc. of the 24th Annual Int'l Symposium on Fault-Tolerant Computing (FTCS), 1994.

[2] Barak, B., O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K.Yang, "On the (Im)possibility of Obfuscating Programs," CRYPTO 2001, Santa Barbara, CA, 19-23 Aug 2001.

[3] Blum, M., W. Evans, P. Gemmell, S. Kannan, and M. Noar, "Checking the Correctness of Memories," Algorithmica 12(2/3):225–244 (1994).

[4] Devanbu, P. and S. Stubblebine, "Stack and Queue Integrity on Hostile Platforms," IEEE Transactions on Software Engineering 28(1):100–108 (Jan. 2002).

[5] McCarthy, J., P. Abrahams, D. Edwards, T. Hart, and M. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Second Edition, 1985.

[6] Menezes, A.J., P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, October 1996.

[7] Merkle, R.C., "A Certified Digital Signature," Proc. of Advances in Cryptology (Crypto '89), 1989.

[8] Schorr, H. and W. Waite, "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," Communications of the ACM 10(8):501–506 (August 1967)

[9] Sun Micro-Systems, Inc, "Javacard 2.0 Application Programming Interfaces," Oct. 1997, (http://java.sun.com/java/products/javacard).

[10] Wang, C., J. Davidson, J. Hill, and J. Knight, "Protection of Software-Based Survivability Mechanisms," Proceedings of the 2001 Dependable Systems and Networks (DSN'01). July, Goteborg, Sweden.

[11] Yee B. and D. Tygar, "Secure Coprocessors in Electronic Commerce Applications," Proc. First USENIX Workshop on Electronic Commerce, July 1995.