A Tamper-Detecting Implementation of Lisp

Dennis Heimbigner
Computer Science Department
University of Colorado, Boulder, CO 80309-0430, USA
Dennis.Heimbigner@colorado.edu

Abstract

An important and recurring security scenario involves the need to carry out trusted computations in the context of untrusted environments. It is shown how a tamper-detecting interpreter for a programming language – specifically Lisp 1.5 – combined with the use of a secure co-processor can address this problem. The term "tamper-detecting" means that any attempt to corrupt a computation carried out by a program in the language will be detected on-line and the computation aborted. This approach executes the interpreter on the secure co-processor while the code and data of the program reside in the larger memory of an associated untrusted host. This allows the co-processor to utilize the host's memory without fear of tampering even by a hostile host. This approach has several advantages including ease of use and the ability to provide tamper-detection for any program that can be constructed using the language.

1. Computing in a Hostile Environment

An important and recurring security scenario involves the need to carry out trusted computations in the context of untrusted environments. One approach is to combine a secure co-processor [6][15] with an untrusted host computer. The secure co-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 co-processor in an attempt to corrupt the operation of the secure co-processor.

This paper demonstrates a solution where a programming language system – specifically Lisp 1.5 – is used to provide a convenient and general mechanism for tamper-detecting utilization of a specific resource, namely the memory of an untrusted host. An interpreter for the language system resides on the secure co-processor, but the programs and data executed by the interpreter reside in the memory of the untrusted host.

In this context, the term "tamper-detecting" means that any attempt to corrupt a computation carried out by a program in the language will be detected on-line (before the computation is complete), and the computation will be aborted.

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 and to demonstrate proof-of-concept. Lisp provides a simple, usable, and complete language. It has a small interpreter [11] that can easily be implemented on a secure co-processor with limited resources. Equally important, Lisp uses lists as its only data structure, both for programs and for data; hence tamper detection can be applied to both code and data with no extra effort. Thus Lisp provides a good platform for exploring issues in tamper-detecting language implementations.

3. Lisp List Representation

This paper assumes familiarity with the Lisp 1.5 language and its implementation. A complete review of the language and its original implementation is available in the "Lisp 1.5 Programmer's Manual" [8] by John McCarthy et al.

Briefly reviewing, Lisp lists are composed of cells linked via pointers. A standard Lisp cell consists of three fields: (1) Car and (2) Cdr pointers to other cells and (3) a flag field indicating properties of the cell. 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". Cyclic lists are allowed, as are lists with common sublists. The standard flags are as follows.

1. ATOM – the cell format is that of an atomic (i.e., non-list cell) value.
2. NUMBER – a subclass of ATOM indicating that this cell holds a numeric value.

3. FREE – this cell is on the *freelist*.
4. MARK, CARCHAIN, and CDRCHAIN – for use during garbage collection (see Section 7).

In the implementation used here, the Car of an atom cell points to the name represented as a list of integers, where each integer is the value of a character in the name. The Cdr field of an atom is used to link it into a list of all known atoms (the OBLIST). We assume that atoms can be created but never destroyed. The root of the OBLIST is one of several special pointers kept in the memory of the secure co-processor where they are immune from insecure modification. The value of a non-integer atom is handled using the traditional ALIST and APVAL mechanisms. Nil is a special atom whose value is itself and which is traditionally used to terminate lists. Numeric atoms contain the integer value in the Cdr field. This makes the reasonable assumption that an integer and a pointer have the same size.

4. Attack and Trust Assumptions

The critical trust assumption is that any values kept in the memory of the secure co-processor cannot be directly read or modified by the untrusted host. Thus the code and data on the secure co-processor constitute the trusted computing base for the Lisp implementation. The only assumed attack mechanism by which the host can tamper with a computation of the secure co-processor is through the values the host returns in response to read requests from the secure co-processor. Specifically not addressed are any physical attacks against the secure co-processor.

5. Secure Co-Processor –Host Access Protocol

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

- *read(i)* – return the content of host memory location *i*.
- *write(i,c)* – write *c* as the new content for host memory location *i*.
- *alloc(n)* – allocate *n* sequentially located cells of new host memory and return the address of the start of that memory.
- *release()* – let the host reclaim all allocated memory.

To simplify the Lisp interpreter in the co-processor, the above operations are wrapped by the following Lisp-oriented operations used during normal computation.

- *CAR(p), CDR(p)* – read the cell (with tamper detection) pointed to by *p* and extract either the Car or Cdr field respectively.
- *CONS(p,q,f)* – construct the content of a cell with *p* in its Car field, with *q* in its Cdr field, and with *f* in its flag field. Then obtain a pointer to a free cell from the *freelist* and write (with tamper detection) the newly constructed cell into that free cell. The *freelist* is a special list of unused cells linked together through their Cdr fields and with the special FREE flag set. Cells 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 to construct a new freelist.

6. Basic Elements of Tamper-Detection

Tamper-detection 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.

Tampering with a cell's content is detected by adding a cryptographic signature as a new field in each cell (the ? field in Figure 1). The signature is computed using any reasonable collision/computation resistant (i.e., one-way and hard to invert) and second pre-image resistant hash function such as SHA-1 [9] or MD5 [9]. The hash function takes the ordered concatenation of the following four values as its input.

- Cell content – the Car, Cdr, and Flags fields (also ordered).
- Cell address – the address from which the cell was read.
- Time stamp – the current epoch index.
- Secret key – a key known only to the secure co-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 ? signature field allows the content of a cell to be validated using only the address of the cell, the content of the cell, and the secret key and epoch index information contained in the secure co-processor. Note that the signature, by itself, does not prevent replay attacks, only synthesis attacks.

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 numbers. This suggests that the need for the epoch index can be replaced by using the secret key instead. This has the advantage of introducing a new secret key for every epoch, which provides a natural mechanism for re-keying. In subsequent discussion, it is

assumed that the epoch index and the secret key are combined into a single *epoch key*.

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 (128 bits). 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.

While the signature prevents synthesis, replay is prevented by enforcing the *write-once-per-epoch* property. This means that during an epoch, any given cell in the host memory will be written at most once. This property is enforced by the fact that the only memory writing that can occur within an epoch is through the CONS operator, which is defined to always store its result in a new cell taken from the freelist.

Within an epoch, a cell will have at most two values as its content. For cells that are already allocated at the beginning of the epoch, their content will never change. For cells that are on the freelist, their initial content is the initial value as a member of the freelist. The second is the value written into it at the time it is allocated. Thus for any cell, the only possible replay attacks are the following:

1. Replay the cell content from another epoch,
2. Replay the content of some other cell in the same epoch,
3. Replay the content of an allocated cell as it was when it was on the freelist.

Since the signature includes the cell address and the epoch key, cases 1 and 2 can be detected by failure to validate the signature when the cell is read from the host processor. Case 3 will be detected by the presence of a FREE flag, which cannot occur when reading a cell reachable by any pointer (other than the head of the freelist). Thus the only cell value that an attacker can return is the correct value of the cell as written (once) during the epoch.

The write-once property has some important consequences. In particular, it disallows use of traditional extensions to Lisp such as REPLACA, REPLACD, and PROG because they support direct cell modifications. Write-once does not prevent lambda binding (using an ALIST) and SETQ (using an APVAL list); see the longer technical report [6] for details.

7. Epoch Transition by Garbage Collection

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

- All unreachable cells have been linked into a single freelist.
- All cell signatures (reachable and unreachable) have been updated based on the new epoch key.

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

In the following discussion, familiarity is assumed with the common approaches to garbage collection. In particular, familiarity is assumed with the standard mark-and-sweep approach, which was chosen because it isolates the collection activity into a single phase for which special anti-tampering mechanisms can be used. Knowledge of the well-known Schorr-Waite(-Deutsch) [12] algorithm for marking is also assumed. This algorithm was chosen 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.

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 co-processor. As each cell is first reached, it is marked. At the point where a cell is touched for the last time in the walk, its content is re-signed using the new epoch key. Thus at the end of the traversal all reachable cells have been touched and re-written. Since they have been re-signed using the new epoch key, 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 in Figure 1 show these two cases respectively.

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.
(2,3) Car or Cdr Chaining – some cells on the current depth-first path will have their Car or Cdr fields 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 MARK flag set.

Figure 1 shows a point in the traversal of a set of lists. The dotted lines indicate the boundary between the secure co-processor and the host. The box labeled P1 at the left represents a root pointer in the secure co-processor. The boxes labeled M1 and M2 represent special pointers in the secure co-processor. They are used

to track the state of the marking procedure. Thus, M1 points to the last cell in the current depth-first path (C4) and M2 points to the next cell to be marked (C5). Note the reversal of several pointers in cells C1, C2, and C4 and the associated flags ? (for Car reversal chaining) and ? (for Cdr reversal chaining).

After marking is completed, the sweep phase examines every cell in the sequential order defined by its memory address (the chunks of *alloc()*'d memory are tracked using a special ALLOC list). If the cell is unmarked, then it is unused, 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 co-processor just inverts the sense of the mark bit so that in the next garbage collection, all cells will be considered unmarked.

8. Tamper Detecting Garbage Collection

The goals for garbage collection are three-fold:

1. Immediately detect attempts to modify a cell's content,
2. Detect replays no later than the end of garbage collection.
3. Detect replays before they can cause garbage collection to fail,

Thus, we are willing to allow replays to occur as long as they do not corrupt garbage collection, but in any case, replays must be detected before normal computation resumes.

The first goal is easily met if we continue to sign our cells every time we write them to the host memory. Again assuming that our hash function is hard to invert, we assume that attempts to modify a cell's content will fail.

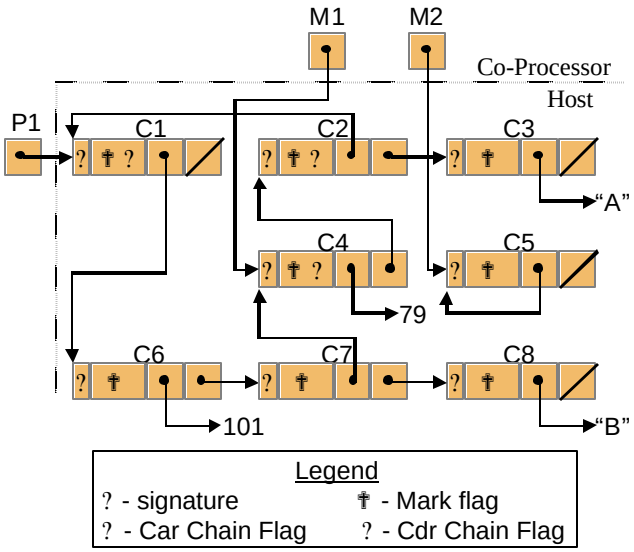

Figure 1. Schorr-Waite marking

So at the start of garbage collection, a new secret epoch key is computed. During garbage collection, ? (the signature field) is recomputed every time a cell is modified. The specific epoch key, new or old, is chosen based on the step in the marking phase.

During Car and Cdr chaining, the cell is re-written using the old epoch key. When traversal of a cell is completed, the last action is to recompute its ? field to contain its signature but using the new epoch key.

When reading a cell, it is verified using the old epoch key if the cell appears unmarked, or appears to be involved in Car or Cdr chaining. Otherwise, it is verified using the new epoch key.

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

If the cell appears to be unmarked then its signature is validated using the old epoch key. 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 signature field of the free cell is computed using the revised content and using the new epoch key.

If the cell appears to be marked, then its signature is validated using the new epoch key, and otherwise it is left untouched.

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

9. Replay Attacks Against Garbage Collection

While a keyed, cryptographically strong hash function prevents the host from synthesizing corrupt cell content, it does not necessarily prevent replay attacks. Whenever a cell is read from the host memory during either the mark or sweep phases, 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 content of the cell.
2. It can replay the content of the cell as it was when involved in Cdr chaining.
3. It can replay the content of the cell as it was when involved in Car chaining.
4. It can replay the content of the cell as it was before garbage collection began.

Obviously Case 1 causes no problem. Cases 2 and 3 can only usefully occur during the mark phase because the sweep phase does not use the Car and Cdr chaining flags.

Even during the marking phase, these cases can only occur when it possible to reach a cell by more than one path. As figure 1 shows, this can occur for cells that are sublists of more than one list (cell C4) or are part of a cyclic list (cell C5). 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

If the host provides case 2 or 3 replay, this can cause no difficulties because 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 content of the cell may indicate the cell has already been marked. If instead the host replays the old, unmarked cell content, then the garbage collector will happily re-mark that cell and everything reachable from it. This will continue as long as the host replays unmarked cell values, possibly forever. This re-marking by itself causes no harm because marking is an idempotent operation.

The same thing may also happen during the sweep phase. That is, the host may replay the unmarked content of each cell. In this case, an active cell will be treated as a free cell and erroneously added to the freelist. Again, this causes no immediate harm since the cells of the freelist are never revisited during the sweep phase.

10. A Counting Solution

The only damaging effects of a Case 4 replay are to cause the mark phase to re-mark cells (possibly endlessly) or to cause the sweep phase to free cells that are really reachable. The effects of both attacks can be controlled using a counting technique.

Assume that at the beginning of garbage collection, we know Tc, the total number of allocated cells. In the untampered state:

$$Tc = Rc + Fc \quad (1)$$

where Rc is the total number of reachable cells and Fc is the total number of free cells. In a possibly tampered state, we have the following inequality.

$$Tc \stackrel{?}{} Mc + Sc \quad (2)$$

where Mc is the number of unmarked cells seen during the mark phase and Sc is the number of unmarked cells seen during the sweep phase. The inequality is a consequence of the following two inequalities.

$$Rc \stackrel{?}{} MC \quad (3) \qquad Fc \stackrel{?}{} Sc \quad (4)$$

These two inequalities come from the following observations. First, if the attacker uses a case 4 replay during the mark phase, it will increase the number of apparently unmarked cells by 1, hence equation (3) will hold. If the attacker uses the same replay during the sweep phase, it will increase the number of apparently free cells by one, hence equation (4) will hold. Note that the number of marked cells can never be falsely increased because that would require the attacker to be able to mark a cell without detection, which is hypothesized to be impossible if our signature function is not invertible.

Using equations (1) and (2), it is then possible to detect the occurrence of tampering no later than the end of the sweep phase of garbage collection, which is our second goal. At that point, the number of marked cells plus the number of free cells will have been counted and if the total is greater than the total number of available cells (Tc), then tampering has occurred.

Our one remaining problem is the possibility of endless replay during the mark phase. If the attacker always replays unmarked cell values, then there is a potential for the mark phase to loop forever in the presence of any cyclic lists. To address this, we note one more equation.

$$Mc > Tc \;\; ? \;\; Tc \stackrel{?}{} Mc + Sc \quad (5)$$

This indicates that if the number of unmarked cells Mc ever exceeds Tc, then equation (2) will of necessity be true and hence tampering must be occurring. Thus if we track the number of unmarked cells read, we are guaranteed to eventually detect a loop and signal a replay attack.

11. Preliminary Performance Measurements

An implementation of the tamper-detecting Lisp system has been completed. Preliminary performance measurements have been collected using g++ version2.95.3 on an Ultra-2 Sparc platform. These measurements must be interpreted carefully because of the platform and because of the following assumptions and implementation limitations underlying those measurements.

The most important limitation concerns signing overhead. We use a public domain, software-only implementation of the MD5 signing code from RSA with an average signing time of about 10 microseconds for signing 512 bits. This means that signing time dominates the current set of measurements and gives the appearance that the cost of tamper-detection is large. Given signing hardware and/or faster signing functions, it should be possible to reduce this cost to, say, 1 or 2 microseconds, at which point the signing cost becomes manageable.

The memory bandwidth between the secure co-processor and the host also impacts the overall performance, but it is not separately modeled in our performance measurements.. Not only is every cell read by the secure co-processor, the cell size increases because of the addition of the signature field. The communication channel between the host and the co-processor should be

something like Firewire, USB 2.0, or a direct PCI bus connection to avoid bandwidth bottlenecks.

With these limitations in mind, Chart 1 shows some preliminary performance measurements for doing the garbage collection mark phase on acyclic graphs whose depth ranges from 10 to 100. This particular measurement is included because tamper-detecting garbage collection, and marking in particular, is critical to the operation of the system. The list graphs to be marked 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 Y-axis of the graph is the log of the elapsed time in microseconds and the X-axis is the graph depth.

The top line represents marking time when anti-tamper is activated. The second line from the top shows the normal marking time. The bottom line is the ratio (about 17), and the second line from the bottom line is the total number of cells in the graph.

In summary, the performance measurements appear to indicate that a feasible tamper-detection Lisp implementation can be constructed if certain bandwidth and signing speeds can be achieved. But detailed and accurate performance measurements must wait until the implementation is re-hosted onto real co-processor hardware.

12. Related Work

The approach proposed in this paper is directly inspired by the prior work in tamper-detecting 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].

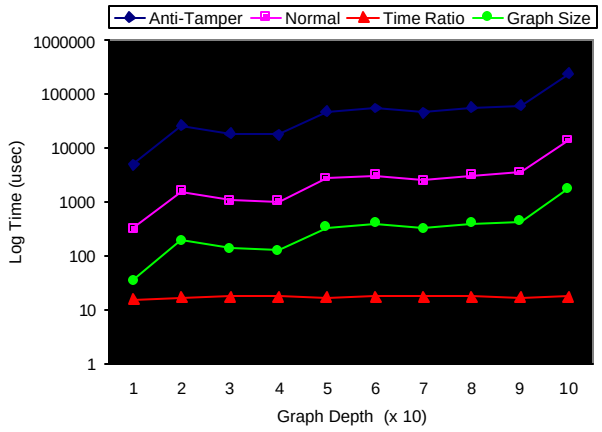

Chart 1. Graph marking times

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-detection. 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. This solution also reduces and simplifies the code that must reside on the host processor. Data structure specific code is not required. Instead, the only required code is that necessary to allow the secure co-processor to read and write the host's memory and to request the allocation of blocks of the host's memory (Section 5). Thus using the language approach, it should be easier to construct programs that can safely avail themselves of untrustworthy host memory.

This work explicitly assumes the use of secure hardware whose memory cannot be read or modified by the untrusted host. There exist software-only solutions [14] to the trusted computing problem that use various forms of obfuscation to prevent an untrusted software program from analyzing the actions of the trusted software. A recent theoretical result [2] casts doubt on the generality of this approach, and indicates that completely general software-only solutions may be impossible.

The use of *cryptopaging* [5][13] is another possible alternative to the approach presented here. Cryptopaging treats the host as secondary memory and the secure co-processor uses it as the target for paging its memory. Replay is prevented by maintaining a complete Merkle hash tree [10] whose leaves are the available pages of the host memory. The nodes of the Merkle tree are also kept in host memory. Cryptopaging has the advantage that it is fast because it uses modified hardware for signing and memory retrieval. This is also a disadvantage since our approach can use off-the-shelf hardware. Its other disadvantage is page-size. Our approach accesses memory in smaller chunks and uses language semantics to ensure that no replay occurs. The cryptopaging approach uses additional memory for storing the Merkle hash tree interior nodes in the host memory. It also must find a satisfactory trade-off between page size and the size of the Merkle tree. An efficient cryptopaging approach also requires a significant degree of locality of reference. Languages like Lisp are notorious for breaking paging algorithms because they rapidly lose any locality of reference properties. Each approach has merits and demerits and a more direct comparison using real hardware would be interesting.

13. Summary

This paper proposes the use of a tamper-detection programming language implementation plus a secure co-processor to achieve trusted computing in an untrusted environment. The claim that the implementation can detect tampering rests on the following three arguments.

1. An attacker (the host) can never undetectably provide corrupt data to the secure co-processor because all cells in memory are signed.
2. Write-one-per-epoch combined with signing of cells guarantees that an attacker cannot successfully replay data during an epoch..
3. Replaying unmarked cell content during garbage collection can be detected using the counting technique.

A preliminary implementation of the tamper-detection Lisp has been completed. Future research will attempt to move this implementation to a true secure co-processor environment. Parallel research will examine possible applications of this technique to more complex RAM programming models.

14. Acknowledgements

This material is based in part upon work sponsored by DARPA, SPAWAR, AFRL, and AFOSR under Contracts N66001-00-8945, F30602-00-2-0608, and F49620-01-1-0282. The content does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. The comments of Professors Devanbu and Pandey of the University of California at Davis and Tom Green of the University of Colorado are also gratefully acknowledged.

15. 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] Gassend, B., D. Clarke, M. van Dijk, S. Devadas, and E. Suh, "Caches and Merkle Trees for Efficient Memory Authentication," Proc. of the 9th High Performance Computer Architecture Symposium (HPCA'03), Anaheim, CA., 8-12 Feb. 2003.

[6] Heimbigner, D., "A Tamper-Resistant Programming Language," Department of Computer Science Technical Report CU-CS-931-02, University of Colorado, 20 May 2002.

[7] IBM Cryptographic Products, "IBM PCI Cryptographic Processor General Information Manual," Sixth Edition, May 2002, (http://www-3.ibm.com/security/cryptocards/html/library.shtml).

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

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

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

[11] Queinnec, C., "Lisp – Almost a whole Truth," Research Report LIX/RR/89/03, École Polytechnique, France, December 1989, pp. 79–106.

[12] 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)

[13] Smith, S., "Secure Coprocessing Applications and Research Issues," Los Alamos Unclassified Release LAUR-96-2805, Los Alamos National Laboratory, August 1996.

[14] 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.

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