A Tamper-Resistant Programming Language System

Dennis Heimbigner

CU-CS-1010-06

June 2, 2006

University of Colorado at Boulder
Technical Report CU-CS-1010-06
Department of Computer Science
430 UCB
University of Colorado
Boulder, Colorado 80309-0430

A Tamper-Resistant Programming Language System

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-resistant interpreter for a programming language – currently Lisp 1.5 – combined with the use of a secure co-processor can address this problem. This solution 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-resistance for any program that can be constructed using the language. The language approach enabled the development of two novel mechanisms for implementing tamper-resistance. These mechanisms provide alternatives to pure Merkle hash trees. Simulated relative performance of the various mechanisms is provided and shows the relative merits of each mechanism.*

1 INTRODUCTION: 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. This problem recurs in a number of contexts. It is desirable, for example, for mobile agents [13,14,28] to act as surrogates for humans and for those agents to be able to engage in sensitive actions such as credit card transactions. The mobile agent is executing on a remote host, and the owner of that host, if malicious, may attempt to tamper with the operation of that agent. Another example involves operation of security infrastructure such as an intrusion detection system [10,16]. It is likely that this infrastructure executes in part on a host that is subject to external attack. The security infrastructure is therefore also subject to attack, and may in fact represent a high profile target for external attacks. For these and other scenarios, executing trusted code in an untrusted environment is an important capability.

One promising approach to trusted execution is to combine a trusted secure co-processor [19,37,38] with an untrusted host computer. The secure co-processor establishes the environment in which to perform trusted computations, while the insecure host provides memory (and other) resources that may be used by the trusted processor. There is no guarantee, however, that the host will not tamper with the content of its memory in an attempt to corrupt the operation of the secure co-processor. The problem to be solved in this context is to allow the co-processor to easily use the host's memory while still being able to detect attempts by the host to tamper with the co-processor's utilization of that memory.

This paper demonstrates a novel solution to the untrusted environment problem by combining a programming language approach and a secure co-processor to provide a convenient and general mechanism for tamper-resistant utilization of the memory of an untrusted host. In this approach, an interpreter for the language executes on the co-processor, and the interpreted program and associated data reside on the untrusted host. The key concept is that tamper-resistance is built into the language interpreter's implementation.

The language approach has several positive characteristics. 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. The only required code is that necessary to allow the secure co-processor to read and write the host's memory and to manage the allocation of blocks of the host's memory. The use of a simplified and common host interface also allows the use of multiple co-processors and the use of multiple languages. Thus using the language approach, it should be easier to construct programs that can safely avail themselves of untrusted host memory.

Two issues arise for this research: (1) can it be implemented and (2) what is the most effective mechanism for achieving tamper-resistance. The first issue was addressed by doing a proof of concept implementation [17] (Section 6). With the feasibility established, the second issue was addressed by exploring several different mechanisms and comparing their performance. This latter work is the primary topic of this paper.

Two new tamper-resistance mechanisms were developed in addition to pure crypto-paging. These two new mechanisms were a direct result of using the language approach. One of these new mechanisms involves inter-twining of tamper-detection with the operation of the language: it provides a new paradigm for amortizing signing costs that differs from that provided by

Merkle hash trees. The second new mechanism involves modifying the crypto-pager to utilize information about the activities of the language implementation. This paper describes the implementation of the language interpreter and the two new mechanisms. Relative performance estimates are also obtained for those new mechanisms and for pure crypto-paging.

2 PRELIMINARIES: DEFINITIONS AND ASSUMPTIONS

2.1 Tamper-Resistance

The term "tamper-resistance" refers to two properties.

1. Detection: Any attempt to corrupt a computation carried out by a program in the language will be detected by the interpreter on-line before the computation is materially affected, and the computation will be aborted.
2. Futility: The cryptographic cost for successfully tampering with a computation must be so high as to make the effort essentially futile. Brute force attacks on the signature are assumed to be computationally infeasible. This is in line with prior related work [12], which assumes the adversary has limited computational power. Information theoretic bounds [1,4] are not considered here.

Note that truly "tamper-proof" computations are essentially unachievable because various denial-of-service attacks are always possible. The host can, for example, always send random data to the secure co-processor to trivially disrupt the computation.

2.2 Secure Co-Processor Capabilities

The co-processor need only have a modest amount of computing power. Sufficient bandwidth between the co-processor and the host's memory is, however, critical. A secure co-processor such as the IBM 4758 [18], for example, can access the host memory at near bus speeds, and can provide a suitable platform for implementing tamper-resistant languages.

The amount of memory on the co-processor is assumed to be fixed and not very large; specifically it is assumed to be too small to execute arbitrary programs written in the interpretive language, which means that some amount of host memory is necessary in order to compute the programs. As is shown in Section 6.1, this has implications for the design of the interpreter.

Superficially, the other obvious hardware choice might be some form of SmartCard such as the Java SmartCard [32]. To date, however, the bandwidth between the card and the host has been severely limited because of the historical use of slower serial protocols. As commercial SmartCard interfaces evolve, the bandwidth will increase, and should soon reach the point where they are feasible alternatives for this research. The primary point is that usable hardware exists now and more will appear in the future.

2.3 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 programming language interpreter. 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. This research specifically does not address the problem of physical attacks against the secure co-processor.

2.4 Secure Co-Processor – Host Access Protocol

The complexity of the interface between the host and the secure co-processor is limited by using the following simple, language independent API.

- void read(Addr a, byte[] buffer) throws Error – fill the buffer with bytes starting at the specified host address.
- void write(Addr a, byte[] buffer) throws Error – write contents of the buffer starting at the specified host address.
- Addr alloc(int allocbytes) throws Error – allocate n sequentially located bytes of new host memory and return the address of the start of that memory.
- void release(Addr a, int len) – let the host reclaim the block of memory starting at the specified address.

2.5 Confidentiality

In order to limit the scope of the problem, only the issue of integrity was addressed in this research; the issue of confidentiality was deferred. It seems reasonable, however, to assume that adding confidentiality is a straightforward application of encryption to the values stored in the host memory, although with some additional performance costs.

2.6 Covert Channels

The existence of covert channels [20] is a possibility with the system proposed here. This is only relevant in the case where code and data confidentiality is being enforced. A client watching the code execution sequence by the interpreter may be able to infer information about the data that the mobile code would rather not reveal. This is a known hard problem, and this research considers the issue out-of-scope.

2.7 Program Correctness

It is important to note that just because a program is tamper-resistant does not mean that it is correct. An incorrect program may still leak information by its actions. Thus, various kinds of program analysis tools [22,27] must be applied to the code to provide assurances that it will not leak information. This kind of analysis is not addressed here.

2.8 Space Versus Time

The amount of host memory required for a given interpreted program varies depending on the specific anti-tamper mechanism used. A semi-space garbage collector, for example, doubles the amount of required memory. Crypto-paging (Section 4) requires extra memory to hold the interior nodes of a Merkle hash tree. It is assumed that host memory is cheap and co-processor memory is expensive. Time costs such as signing are assumed to be the critical performance metric. Thus, the amount of memory used by each mechanism is essentially ignored in the following discussions.

3 TAMPER-RESISTANCE MECHANISMS

The basic approach to a tamper-resistant language implementation involves two elements. First, the core mechanism for achieving tamper-resistance is to cryptographically sign the run-time data structures and code of the programming language so that any direct attempt to change them can be detected by the interpreter. The signature is computed using any reasonable one-way and hard to invert hash function such as AES [25].

The second element of tamper-resistance is to prevent replay. Replay occurs when the content of some memory cell has been written multiple times by the co-processor, and then the co-processor asks the host for the current content of the cell. If the host is malicious, it can return any of the values ever written in that cell, not just the most recent, and the co-processor has no way to detect that fact. Thus, the host may be able to disrupt the computation without being detected. Note that signing (or even encrypting) the memory content alone will not prevent replay. The focus, then, must be on preventing replay; signing will suffice to cover other forms of tampering.

The mechanisms used here to provide a tamper-resistant language implementation build on a number of well-known existing techniques: two-level paging and Merkle hash trees. A novel capability that derives from the use of a programming language approach, however, exploits the semantics of the language implementation to provide alternatives that can improve upon these existing techniques.

This paper explores three alternative mechanisms for implementing language tamper-resistance. These three are (1) crypto-paging, (2) semantic-paging using Epochs, and (3) a combination of crypto-paging plus semantic-paging. The term “semantic-paging” refers to the use of the programming language semantics to inform the tamper-resistance mechanism.

All three mechanisms build on a two-level paging model. This model treats the co-processor’s memory and the host’s memory as a traditional two level memory controlled by paging. That is, whenever a cell is read from the host’s memory, the whole page in which it resides is read into the co-processor’s memory. Traditional paging caches, working sets, and page aging algorithms [11] can be employed to avoid always re-reading the host memory.

4 CRYPTO-PAGING

Crypto-paging [15,30] is a well-known and straightforward approach to managing replay. In its simplest form, each page in host memory has a corresponding cryptographic signature stored in the protected memory of the secure co-processor. In effect, the co-processor stores the page tables plus signatures for each page. Every read of a page is re-validated by computing the signature and comparing it to the stored signature. Similarly, writing a modified page also changes the protected signature. Note that replay protection requires that the crypto-paging system store all signatures in the secure co-processor. Since the page size may be larger than the input block used by the hash function, chained block hashing may be used to compute a signature for large pages.

This approach generally requires significant memory in the co-processor to hold the pages and their signatures. The memory usage can be varied by using a Merkle hash tree [24] with varying degrees of tree height. Minimal co-processor memory usage occurs when the co-processor stores only the root signature of the Merkle tree – plus a few data pages. The most memory is used if

the tree is flattened completely, which corresponds to the simple case above where each host page signature is kept in the co-processor memory. Intermediate degrees of tree height use intermediate amounts of memory. The cost, of course, is that increased tree height increases the average access time up to O(log(n)), where n is the number of host memory pages.

Crypto-paging is non-semantic because the algorithm does not care why a given page is read or written or in what order, nor does it track what parts of the page are modified beyond the fact that the page was modified at all. It is thus simple to implement and can be used for any purpose.

5 SEMANTIC PAGING USING EPOCHS

As part of this research, a new mechanism for replay-detection was developed that exploited the run-time semantics of the programming language, and specifically exploited run-time garbage-collection.

This approach divides the whole computation into a series of periods (the epochs). The sequence of epochs continues until the computation is complete. The key idea is that during an epoch, the content of the host memory is never written more than once. This property is referred to as write-once-per-epoch. During the move from one epoch to the next the host memory is garbage collected and the next epoch begins.

The epoch approach is interesting because it amortizes the costs for preventing replay in a way that is different than Merkle-based crypto-paging. The crypto-paging approach pays the cost of managing replay every time the memory is written. In the epoch approach, all of that cost is delayed until the end of the epoch. The epoch model does, however, require some kind of periodic memory reorganization to be part of the language runtime.

6 TAMPER-RESISTANT LISP 1.5

In order to make the semantic paging approach concrete, a specific language – Lisp 1.5 – was chosen as the target. It was chosen primarily for its simplicity and to demonstrate proof-of-concept. Lisp provides a simple, usable, and complete language. It has a small interpreter [23] 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-resistant language systems.

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” [23] by John McCarthy et al.

Briefly reviewing, Lisp provides a single data structure: lists. Lists are used to represent all data structures. They are also used to represent Lisp programs. These lists are composed of cells linked via pointers. A standard Lisp cell consists of three fields: (1) *car* and (2) *cdr*, which are pointers to other cells, and (3) a *flags* 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 flags are used to identify specialized classes of cells, including the following.

- Atoms – an atom cell is the head of a specialized list structure representing an atomic (i.e. non-list) value.
- Number – a subclass of Atoms indicating that this cell holds a numeric value.
- Free – this cell is unused and can be claimed by the cons operator.

To simplify the Lisp interpreter in the co-processor, the following cell-oriented operations were implemented on top of the host-interface operations listed in Section 2.4.

- public void readCell(Addr a, Cell c) throws Error – read cell at the specified address.
- public void writeCell(Addr a, Cell c) throws Error – write cell at the specified address.

6.1 Interpreter Organization

The Lisp interpreter that executes on the secure co-processor is designed to use a strictly bounded amount of co-processor memory. This reflects the limited size and memory of the secure co-processor. This poses problems when the bounded interpreter must execute a Lisp function that may be recursive and may require a stack of arbitrary depth. The solution we adopt here is to implement the interpreter using a continuation-based architecture [26]. Simplifying, the interpreter consists of a loop that removes the top action off of the continuation stack. It decomposes that action into a bounded part and a remainder part. It pushes the remainder part back onto the continuation stack and executes the bounded part. This cycle continues until the continuation stack is empty. For a description of the practical use of continuations, refer to the Scheme programming language [31].

As the interpreter performs its computations, it produces intermediate results. The reverse function for example accumulates the reverse list as it walks the original list. These intermediate results must be visible to the garbage collector so that they will not be marked as unused. This is accomplished by forcing the interpreter code to use a stack of cell pointers in host memory for saving its intermediate results. This is simplified by providing a macro system for Java that hides the saved-value stack.

The page cache page size is defined to be some integral number of Lisp cells. The cache then keeps some fixed number of pages. The page replacement algorithm is essentially LRU except when otherwise dictated by semantic information. This is more or less an arbitrary choice since good algorithms for Lisp are difficult to define.

7 SEMANTIC TAMPER-DETECTION FOR LISP

The semantic tamper-detection mechanism exploits the semantics of Lisp and its implementation in order to detect tampering. In this model, each cell has a *signature* field in addition to its *car*, *cdr*, and *flags* fields. The *signature* field of a cell is a computed cryptographic hash of the ordered concatenation of the following components:

- Cell content – the *car*, *cdr*, and *flags* fields (also concatenated),
- Cell address – the address from which the cell was read,
- Secret key – a key known only to the secure co-processor and recomputed periodically.

Whenever a cell is read from the host, the signature is recomputed and if it matches the stored signature, then it is assumed that the *car*, *cdr*, and *flag* fields are valid. This provides basic protection against synthetic attacks.

Replay detection is implemented using epochs. The whole computation (the program execution) is divided into epochs. The secret key used in signature computations is associated with epochs and is recomputed at the end of each epoch. The sequence of epochs continues until the computation is complete.

Replay is prevented within an epoch 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 newly allocated cell.

Within an epoch, a cell will hold 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 initially unallocated, their initial content is a special value. At the time of allocation, such cells will be written with the second value. 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 unallocated.

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. 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 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) since new bindings at the front of the list will override older bindings further down the list.

8 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.

1. All unreachable cells have been marked as free.
2. All cell signatures (reachable and unreachable) have been re-keyed based on a new secret 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 may be written several times. The next sections discuss how to maintain replay protection during garbage collection.

The following discussion first addresses the general approach to detecting tampering during garbage collection. Familiarity is assumed with the common approaches to garbage collection in which the collection activity is isolated into a single phase for which special anti-tampering mechanisms can be used. In Section 10 a generic mechanism is defined for two pass collectors that provides a simple test for tampering. Then the generic mechanism is extended to two specific collectors: mark-and-sweep (Section 11) and semi-space (Section 12).

9 TAMPER-DETECTING GARBAGE COLLECTION

The tamper-detection goals for garbage collection are three-fold:

1. Immediately detect attempts to modify a cell's content (synthetic attacks);
2. Immediately detect replays that may cause garbage collection to fail;
3. Detect all other replays no later than the end of garbage collection.

A bounded amount of replay can be allowed to occur as long as it does not corrupt garbage collection. In any case (goal 3) all replays must be detected before normal computation resumes.

The first goal is easily met if cells continue to be signed every time they are written to the host memory. Again assuming that the hash function is hard to invert, attempts to modify a cell's content will fail. At the start of each 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 the mark phase, the cell is re-written using the new epoch key.
- When reading a cell, it is verified using the old epoch key if the cell appears unmarked. 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 marked as unallocated. 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. 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 flagged as unallocated 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.

Goal 2 is garbage-collector specific. Goal 3 can be provided in a manner partially independent of the garbage collector.

10 COLLECTOR INDEPENDENT SEMANTIC REPLAY DETECTION

Many collectors have two phases: (1) a mark phase (i.e. walk and mark all allocated cells) and (2) a sweep phase (for cleanup). For such collectors, it is possible to collect information that will reveal the occurrence of replay.

During the mark phase, a malicious host has the ability to return two classes of values when any cell is read. It may return the original, unmarked value, or it may return some marked value stored during the mark phase.

Suppose our collector counts the following information.

• M_m – the number of cells marked during the mark phase.

• M_s – the number of marked cells read during the sweep phase.

If no malicious actions occur, then $M_m = M_s$.

If during the mark phase, a malicious host returns the unmarked value for a cell that has already been marked, then the effect is to increase the number of apparently marked cells (M_m) because the mark phase will attempt to mark every cell it reads that is unmarked. Note that the malicious host can not increase the actual number of marked cells because this would require that it synthesize a cell with the mark flag set. So during the mark phase, the number of cells marked (M_m) can be greater than the actual number, but never less.

Analogously during the sweep phase, the malicious host can also replay the initial unmarked value of a cell. In this case, however, the effect is to decrease the number of apparently marked cells (M_s) seen by the sweep phase. The host cannot increase the number of marked cells because it cannot synthesize a cell with the mark flag set.

The net effect is that if replay occurs, then the following must true:

$$M_m' > M_m \text{ and } M_s' < M_s \Rightarrow M_s' < M_m'$$

where M_m and M_s represent the number of marked cells detected if replay did not occur, and M_m' and M_s' represent the number of marked cells detected if replay occurred.

It should be the case that $M_s = M_m$. This means that it is possible to detect almost all replays by counting M_m and M_s and if they differ, then replay must have occurred and the computation can be aborted. This solves goal 3 above.

The following two sections describe the details of addressing goals 2 and 3 for mark-and-sweep and semi-space garbage collectors.

11 MARK-AND-SWEEP GARBAGE COLLECTION

The mark phase is assumed to use the Schorr-Waite algorithm [29]. This algorithm is used here because it avoids the need for a separate stack. It traverses the graph of cells reachable from a defined set of root pointers kept in the secure co-processor. As it performs its depth-first walk, this algorithm temporarily reverses the list structure of the lists on the current path of the walk. As each cell is first reached, it is marked and re-signed using the new epoch key. 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.

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.

(4) Complete – any cell for which *car* and *cdr* chaining is completed; the MARK flag is set.

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 unused, and it is marked as a free cell and is linked to a 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.

11.1 Replay Attacks Against Mark-and-Sweep

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 or *car* chaining.
3. It can replay the content of the cell as it was before garbage collection began.

Obviously Case 1 causes no problem. Case 3 is addressed by the generic tamper-detection described in Section 10. Note that the garbage collector will happily re-mark a cell if it comes from replaying an old, unmarked cell. 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 and hence satisfies our goal 2 from Section 9.

In order for case 3 to cause infinite looping, it has to continue to feed unmarked cells, which can only occur if there is a cyclic list structure. This case can be detected by simply keeping a count of the number of cells that are marked. If this count reaches the total size of allocated memory, then marking stops and replay is signaled.

Case 2 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. In any case, if the host provides case 2 replay then 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

12 SEMI-SPACE GARBAGE COLLECTION

In order to demonstrate the generality of the semantic approach, it was applied to a second form of garbage collector, namely the semi-space garbage collector. A semi-space collector divides the available memory into two equal (sub-)memories. At any point in time, new cells are allocated from one of the two memories. When that memory is exhausted, all active cells are copied and compacted into the other – target – memory. The part of that memory not used by active cells is then available for allocating new cells.

The compaction algorithm is based on using a breadth-first walk of the active cells. The target memory is used as a queue for holding intermediate results during the walk. As each cell is moved to the target memory, its old location is re-rewritten to provide a forward pointer to its new location. During the walk, any reference encountered that points to a forwarding value is replaced by the forward address. Forwarding is indicated by using the mark bit and storing the forward pointer in the *car* field.

Normally, semi-space collection does not require a sweep phase. In order to detect replay, however, it is necessary to walk the abandoned space to get a count of the actual number of marked cells. This allows the application of the generic tamper-detection rule of Section 10. It thus represents an overhead stemming purely from the need to avoid replay.

12.1 Replay Attacks Against Semi-Space Collection

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 three values during garbage collection.

1. It can return the correct contents of the cell.
2. It can return an enqueued cell from the target memory with its original *car* and *cdr* values instead of the forwarded values.
3. It can return the unforwarded contents from the current memory instead of the forwarded contents.

Case 1 is not a problem. Case 2 cannot actually occur because an enqueued cell will be read only once when it is at the front of the queue hence there will be no replay opportunity. It will be written once immediately after.

If case 3 occurs, then the collector will enqueue the cell two (or more) times. There is a limit to this process since it will eventually fill all of the target space, at which point replay can be signaled. The most difficult situation occurs if the replay is carried out some small finite number of times. It means that each copy may have cells pointing to it. Again, this will not cause the collector to fail (goal 2), but it will obviously cause an inconsistency in the target memory.

The following steps will detect this inconsistency at the end of collection.

1. During mark phase, the number of forwarded cells is counted.
2. After enqueuing is completed, a second pass is made over the just abandoned space to obtain a separate count of the number of forwarded cells. If the two counts differ, then we have tampering because of case 2.

This works because we are applying our generic tamper detection rule (Section 10).

Tampering during the mark phase can increase the apparent number of forwarded (i.e. M_m) cells, but can never decrease it. The second sweep forces the tampering host to decide what value to report for each cell. If it reports a truly forwarded cell as not forwarded then this decreases the count of forwarded cells (M_s) but can never increase it. If the two counts differ ($M_s < M_m$) then tampering has occurred.

13 SEMANTIC CRYPTO-PAGING

The second new mechanism for implementing tamper-resistance combines crypto-paging with the semantic approach. This approach informs the crypto-pager that the language system is performing various operations that can help the crypto-pager to avoid unnecessary cryptographic operations and can cause the pager to perform additional actions as it operates.

For this research, two optimizations were used.

- Re-keying: Re-keying is essential to tamper detection because it reduces the period of time in which an attacker can attempt to break the cryptographic signature. This is especially important when using short signatures to save memory. The pure semantic approach naturally incorporates re-keying as part of the epoch transition process. Re-keying with pure crypto-paging requires a separate periodic pass over memory. The semantic crypto-paging approach modifies the crypto-pager to so that it is aware of when the language implementation is about to perform a garbage collection. This tells the pager that a sweep of memory is about to occur and that it can carry out re-keying as the sweep occurs. Thus, the extra sweep is avoided.
- Abandonment: This optimization is specific to compacting collectors. The pager is informed that some portion of memory no longer contains information of interest. During the next collection cycle, the pager can use a form of “zero-on-read” to avoid computing a signature for a page from the abandoned space.

An additional look-ahead optimization is possible. The pager can be informed that the garbage collector is about to perform a linear sweep of some section of memory. The pager can utilize parallelism to reduce the page loading time. This particular optimization was not implemented because the simulator does not support parallelism. It would, however, be a high payoff optimization when using a real secure co-processor.

14 PERFORMANCE MEASUREMENTS

Access to a real secure co-processor turned out not to be practical for cost reasons, so processor-based timing measurements were not possible. The alternative taken was to run the interpreter and capture the count of three operations under the assumption that the time for any real system would be dominated by these operations.

- The number of pages read from host memory,
- The number of pages written to host memory, and
- The number of hash functions computed; for block-chaining, the total number of hash computations was counted.

Converting these number to real time values is difficult because it depends on at least the assumed times for signing, the bandwidth between the host and the co-processor and the speed of the co-processor.

The goal for the performance experiments was to determine the relative performance of the three tamper-resistance mechanisms. To test this, a tamper-resistant Lisp implementation was written in Java 1.5 and a number of experiments were used to obtain performance information.

A number of variables can affect performance for this system.

- Pager
- Test algorithm
- Available cells
- Garbage Collector
- Page cache size
- Signature cache size
- Cells/page
- Cell size
- Total memory size
- Page size
- Number of garbage collections
- Compact

In order to compare the pager algorithms, it was necessary to fix various values in a way that allowed meaningful comparison. It was decided that all pagers should operate with the same number of cells per page and the same number of available cells, where available cells refers to the space of free cells. This meant that the total amount of memory and/or cell sizes would differ between pagers. Semantic paging, for example, would use a larger cell size, while crypto-paging would have additional space for the Merkle hash tree. Semispace collection would actually have twice the total space of marksweep. However in all cases, the cells per page and the available cells would be constant for all pagers. This is in line with the assumption (Section 2.8) that space on the host processor is not a primary cost. In order to further simplify the state space, two other variables were held constant; the page cache size was fixed at eight pages, and the signature cache size was fixed at two pages. Of course, the latter only was relevant when crypto-paging was being used.

14.1 Experiment 1

The goal of this experiment was to test the garbage collector performance on large random graphs. A special program was constructed that could randomly synthesize graphs of varying depth, width and total number of cells. In all cases, the graph was constructed to consume 80% of available cells. The random graph could either be constructed as “compact” or “scattered”. Compact means that the nodes of the graph are allocated in consecutive memory cells. Scattered means that the graph cell locations are chosen randomly, so that nodes linked together are not necessarily close together physically.

Figures 1a and 1b demonstrate the performance of marksweep versus semispace collection on the random graph when compact is true. The x-axis for both graphs is the number of cells divided by 1024, so the graph runs from 4096 to 32768 available cells. The y-axis is the number

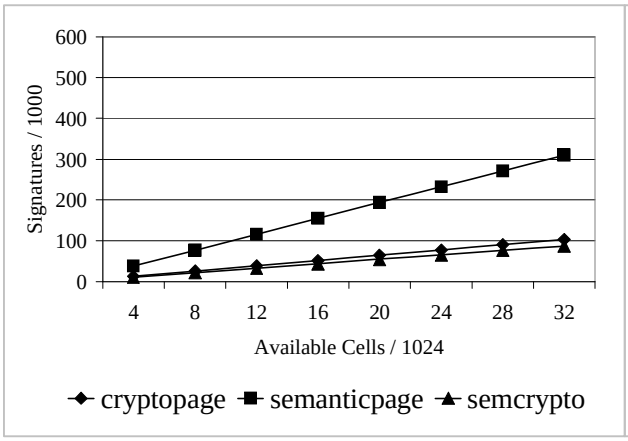

Figure 1a. Compact, Marksweep, Cells/Page=16

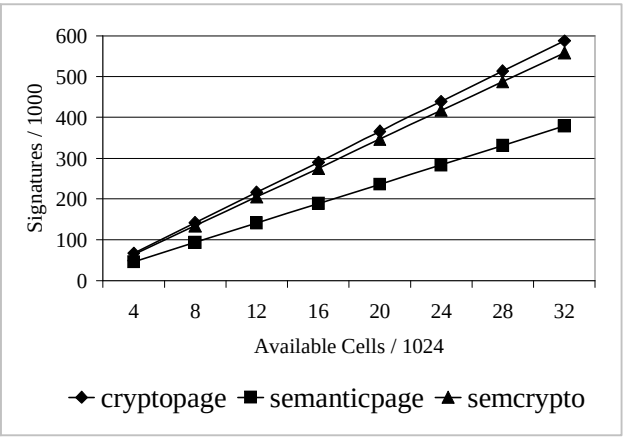

Figure 1b. Compact, Semispace, Cells/Page=16

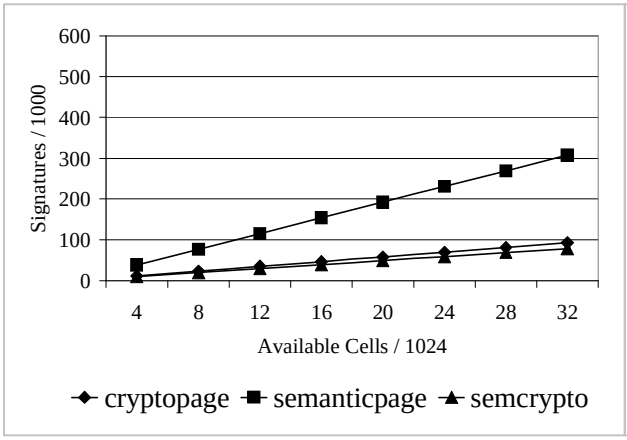

Figure 1c. Compact, Marksweep, Cells/Page=32

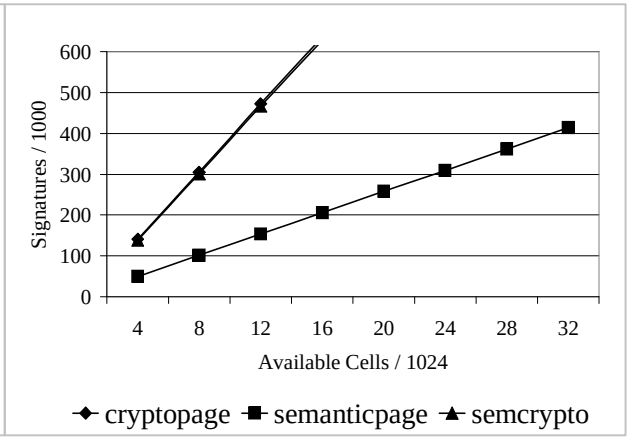

Figure 1d. Scattered, Marksweep, Cells/Page=16

of signatures divided by 1000, so it ran from 0 to 600,000 signings. In Figure 1a, it can be seen that both crypto-paging and semantic crypto-paging performed better than semantic paging by almost a factor of three. When semispace was used (Figure 1b), the cost of semantic paging was only slightly more than in Figure 1a. The performance of the crypto-pagers deteriorated significantly and was worse than semantic paging by a factor of 1.5.

In order to test the effect of cells per page, the test used in Figure 1a was also run with double the cells-per-page and using a compact graph. This is shown in Figure 1c. The two graphs are more or less identical, which indicated that cells-per-page was not a major factor for this experiment.

A fourth test (Figure 1d) demonstrated the effect of compact versus scattered. It repeated the experiment in Figure 1a but using a scattered graph allocation. Again, semantic paging changed slightly, but crypto-paging deteriorated substantially. This is because of the random allocation. When the semantic pager reads a page, it will only sign those cells in the page that it touches. Crypto-paging must re-sign the whole page. If the percentage of cells of interest is small, as it should be using scatter, then crypto-paging requires significantly more signing overhead.

14.2 Experiment 2

The goal of experiment 2 was to test the three mechanisms against the two garbage collection algorithms on an actual Lisp program. The program chosen was the Wang theorem prover from the Lisp 1.5 manual [23]. Executing the prover once did not exercise the collectors sufficiently. So the program was executed multiple times until a pre-defined number of garbage collections had occurred.

Figure 2a shows the cost using marksweep. Crypto-paging out-performed semantic paging by a factor of two and semantic crypto-paging out-performed pure crypto-paging by about 10%. Figure 2b shows the cost using semispace. Note again that semantic paging costs changed the least, while the two crypto-pagers generally performed worse than semantic paging. Note that the semantic crypto-paging line is almost flat and is substantially better than pure crypto-paging, especially compared to Figure 2a. This is a clear demonstration of the effects of the optimizations described in Section 13. Pure crypto-paging is forced to sign pages read from the abandoned space. In addition, it must re-key the total space, which is more than twice the size of available memory. The second interesting measurement is that semantic paging performs well.

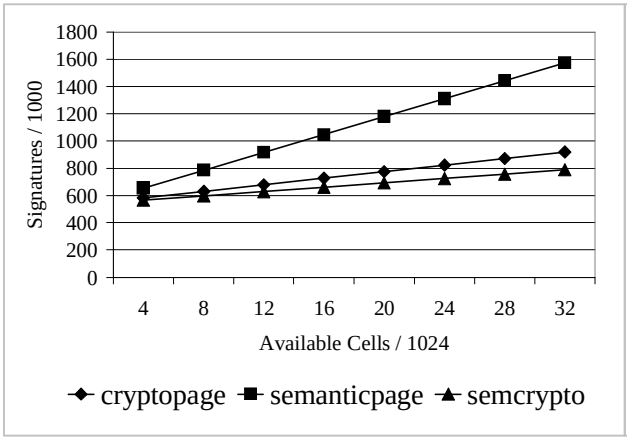

Figure 2a. Marksweep

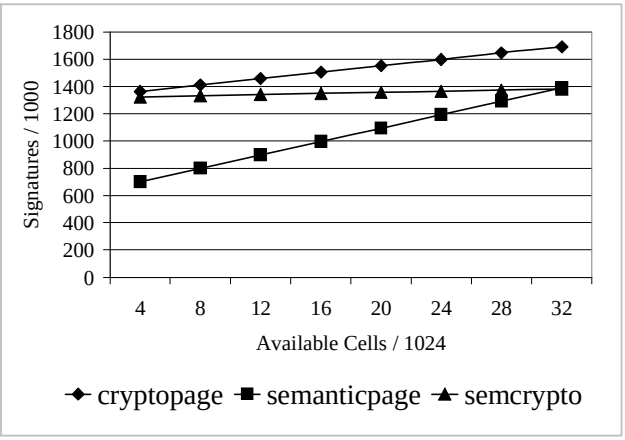

Figure 2b. Semispace

The reason is that the average number of cells read per-page varies by nearly a factor of two between Figure 2a and Figure 2b. For example, at the last data point the number of cells read is 6 for crypto-paging in Figure 2a and is 2 for crypto-paging in Figure 2b. Crypto-paging is sensitive to this value because the lower the number, the more wasted signings are performed by crypto-paging. Semantic paging is much less affected by this number, hence the difference between the graphs.

14.3 Experiment 3

The goal of experiment 3 was to test the effect of varying the number of cells per page. Experiment 1 indicated that it had little effect in the compact random graph case. Here the theorem prover was used as the test program, and the marksweep collector was used. Figures 2a, 3a, and 3b show this test when the cells-per-page is 16, 32, or 64 respectively. The effect seems clear: semantic paging costs are essentially constant, reflecting the fact that it only touches interesting cells. The cost for the crypto-pagers increases as the cells per page increases. This is because they must pay the price of signing the whole page independent of how much of the page is of interest. Again, as with the other experiments, semantic crypto-paging always out-performs pure crypto-paging.

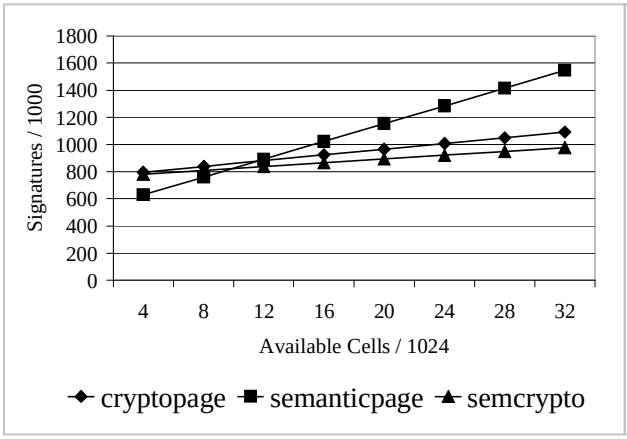

Figure 3a. Cells/Page=32

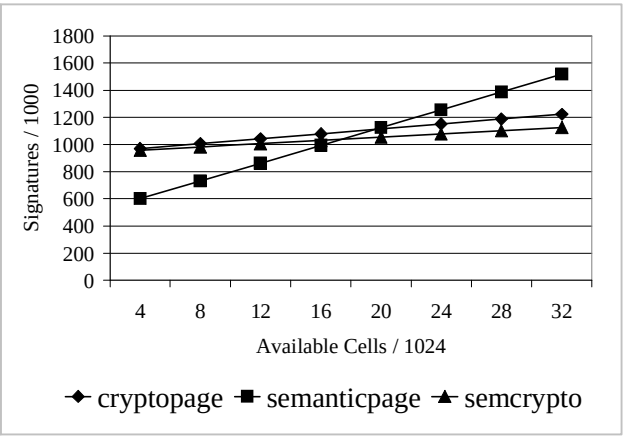

Figure 3b. Cells/Page=64

14.4 Performance Summary

The experiments indicate that the when the data being accessed has a high locality of reference, then the crypto-pagers will probably out-perform the semantic pager. But when the data is more randomly distributed across memory, then semantic paging may be better than either of the crypto-pagers. Perhaps not surprisingly, combining semantic knowledge with crypto-paging produced a mechanism that performed better than pure crypto-paging in all cases.

15 RELATED WORK

Currently, there are two primary approaches to achieving trusted computing in an untrusted environment: software-only approaches based on obfuscation and approaches based on hardware.

The software-only solutions to tamper-resistance are based on obfuscation [6-9,28,36]. The idea is to modify the code of an application in such a way as to prevent an untrusted software program from analyzing the actions of the trusted software. A recent theoretical result [3] casts doubt on the generality of this approach, and indicates that completely general software-only solutions may be impossible. This should not be read as implying that obfuscation is not useful. Rather it means that such approaches must address their limitations with respect to, for example, the time it would take to break the obfuscation. A sufficiently long period may be good enough for practical purposes.

Obfuscation techniques have so far focused solely on protecting the code and making hard for an attacker to reverse engineer it. With one exception [8], however, it does not appear as if obfuscation has been applied to data structures. It is true that obfuscating the code, by re-ordering for example, may affect the order in which the data structure is constructed, but it will not affect the final shape of the data structure. This leaves such structures open to attack. The programming language approach addresses this problem because all of the anti-tamper mechanisms can be applied to data as well as code.

A variety of hardware solutions have been proposed (and in some cases implemented) that are at least tangentially related to this proposal. Encryption of bus traffic [19,21,34] or encryption of memory contents [35] has been proposed to prevent some kinds of tampering. Its advantage is, of course, that it is implemented in hardware and so has a performance advantage. Its disadvantages are (1) it invokes encryption all the time rather than only when needed, and (2) by itself it does not address the replay problem, although it may in practice make effective replay difficult.

An alternative [15] has been proposed that implements a Merkle hash tree [24] in hardware. The leaves are the available pages of the host memory. Except for the root, the nodes of the Merkle tree are also kept in host memory, which represents an unavoidable memory overhead. Merkle memory can be implemented with specialized hardware for signing and memory retrieval, which could provide a significant speed advantage over other approaches. This approach addresses replay, but again uses encryption or signing even when not required. Further, the language-specific semantic optimizations used in this paper cannot be implemented.

Another solution involves executing the trusted code wholly within the secure co-processor's memory. This is the typical solution used with smart cards. JavaCard [32], for example, provides a version of Java in which code and data reside entirely on the smart card. The obvious problem is that the resources of the smart card are limited compared to those of the host. Over time, this limitation will become less important. Another issue is complexity. Executing multiple pieces of

trusted code requires some form of operating system, and requires isolation of the various pieces of code from each other. This introduces significant complexity into the smart card and in effect repeats the untrusted environment problem at a different level.

The approach proposed in this paper and in prior work [17], was directly inspired by the prior work in 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 [4], simple linear lists [1,4], and stacks and queues [4,12].

The language approach has several advantages compared to the data structure approach.

- It is more general since any data structure that can be implemented in the language can be used.
- It hides the complexity of tamper-detection.
- It reduces and simplifies the code that must reside on the host processor; 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.

Using the language approach makes it easier to construct programs that can safely avail themselves of untrustworthy host memory.

16 FUTURE WORK

The most important next step is to re-implement this system using a real secure co-processor. This will provide additional information about performance which should lead to additional research into improving that performance.

Further research into combinations of mechanisms is also clearly warranted. One interesting target is to utilize the different access patterns of code and data. Most interpreters produce separate code segments containing byte-codes. The Lisp 1.5 interpreter used here did not do that. Since code is generally read-only, it may be desirable to apply simpler anti-tamper mechanisms to the code and reserve the more comprehensive mechanisms for modifiable data.

This research is part of a larger program to find novel uses for secure hardware. The clear cost with the solution proposed here is that it requires clients to have a secure co-processor attached to the client host. But less costly alternatives to secure co-processors are possible.

New multi-core versions of processor chips will soon be available. Dedicating one of these cores to security will become an option, especially when the Trusted Computing Association (TCP) [2,5] standard also becomes ubiquitous. It appears feasible to use these new chips to achieve the equivalent of a co-processor/host combination by using one processor in a multi-core processor configuration as a surrogate for the secure co-processor.

SmartCard technology continues to progress. Eventually it will reach a point where it can also serve as an alternative to secure co-processors for the basis for this research.

Finally, a new version of the tamper-resistant language system is being designed that supports a more traditional programming language such as Javascript [33]. This will introduce new opportunities to explore additional semantic optimizations for tamper-resistance.

17 SUMMARY

This paper demonstrates a novel approach to achieve trusted computing in an untrusted environment using a tamper-detection programming language implementation plus a secure co-processor. This solution is simple to use and requires only a limited set of primitives to define the interface between the host and the co-processor.

Three mechanisms were used to support the tamper-resistance. One is crypto-paging (based on Merkle hash trees). The second and third approaches were developed directly as a result of using the programming language approach. The first, the semantic approach, utilized the semantics of the programming language implementation. The second new approach extended crypto-paging to utilize knowledge of the language system to avoid unnecessary operations.

All three mechanisms were implemented for an interpreter for the Lisp 1.5 programming language. Simulated performance experiments demonstrated that the pure semantic approach performs best when data is randomly distributed over the host memory. The crypto-pagers were better when there was significant locality of reference. In all cases, the semantic crypto-pager out-performed the pure crypto-pager.

Acknowledgments

This material is based in part upon work sponsored by DARPA, SPAWAR, AFRL, AFOSR, and ARO under Contracts N66001-00-8945, F30602-00-2-0608, F49620-01-1-0282, and DAAD19-01-1-0484. The content does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

18 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] AMD Corp, "AMD Platform for Trustworthy Computing", Windows Hardware Engineering Conf. (WinHec), May 2003 (http://www.microsoft.com/winhec/papers03.mspx).

[3] 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.

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

[5] Chan, D., Trusted Computing Platform Main Specification Version 1.1b. Feb. 22, 2002 (https://www.trustedcomputinggroup.org/downloads/tcg_spec_1_1b.zip).

[6] Collberg, C., The Obfuscation and Software Watermarking Home Page (http://www.cs.arizona.edu/~collberg/Research/Obfuscation/Resources.htm).

[7] Collberg, C. and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," IEEE Transactions on Software Engineering 28(8):735-746 (August 2002) (http://citeseer.ist.psu.edu/collberg02watermarking.html).

[8] Collberg, C., C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," Int'l Conf. on Computer Languages, 1998, pp. 28-38 (http://citeseer.ist.psu.edu/collberg98breaking.html).

[9] Collberg, C., C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[10] Denning, D., "An intrusion-detection model," IEEE Transactions on Software Engineering 13(2):222-232 (Feb. 1987).

[11] Denning, P., "The working set model for program behavior," Communications of the ACM 11(5):323-333 (May 1968).

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

[13] DOD, "Policy Guidance for Use of Mobile Code Technologies in Department of Defense (DoD) Information Systems," Assistant Secretary of Defense Memorandum, November 7, 2000 (http://www.c3i.osd.mil/org/cio/doc/mobile-code11-7-00.html).

[14] Fuggetta, A., G. P. Picco, and G. Vigna, "Understanding Code Mobility," IEEE Transactions on Software Engineering, 24(5): 342-361 (May 1998) (http://www.cs.ucsb.edu/~vigna/listpub.html).

[15] 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.

[16] Heberlein, L., G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, “A Network Security Monitor,” Proc. IEEE Symposium on Research in Security and Privacy, May 1990, pp. 296-304.

[17] Heimbigner, D., “A Tamper-Detecting Implementation of Lisp,” Proc. of the 2003 Int'l Conf. on Security and Management, Las Vegas, NV, June 2003 (http://www.cs.colorado.edu/users/dennis/publications/heimbigner-sam03-published.pdf).

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

[19] Kuhn, M., “The TrustNo1 Cryptoprocessor Concept,” Purdue University Technical Report CS555, April 1997 (http://www.cl.cam.ac.uk/mgk25/).

[20] Lampson, B. W., “A Note on the Confinement Problem,” CACM 16(10): 613 - 615, October 1973.

[21] Lie D., C. A. Thekkath, and M. Horowitz, “Separating Protection and Resource Management in Operating Systems,” Stanford University VLSI Research Group Report (http://citeseer.ist.psu.edu/lie02separating.html).

[22] Madou, M., B. Anckaert, B. De Sutter, and K. De Bosschere, “Hybrid static-dynamic attacks against software protection mechanisms,” Proceedings of the 5th ACM Workshop on Digital Rights Management, Nov. 2005, pages 75-82.

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

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

[25] NIST, “Announcing the ADVANCED ENCRYPTION STANDARD (AES),” Federal Information Processing Standards Publication 197, November 26, 2001 (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf).

[26] Queinnec, C., “Lisp - Almost a whole Truth,” Research Report LIX/RR/89/03, École Polytechnique, France, December 1989, pp. 79-106.

[27] Rosen, B., “Data flow analysis for procedural languages,” Journal of the ACM, 26(2):322 344, April 1979.

[28] Sander, T. and C.F. Tschudin, “Protecting Mobile Agents Against Malicious Hosts,” Lecture Notes in Computer Science, Volume 1419, 1998 (http://citeseer.ist.psu.edu/sander98protecting.html).

[29] 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)

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

[31] Steele, G. L., Jr. and Gerald Jay Sussman, “The Revised Report on Scheme, a Dialect of Lisp.” MIT AI Memo 452, MIT, Jan. 1978.

[32] Sun Micro-Systems, Inc, “JavaCard 2.0 Application Programming Interfaces,” Oct. 1997 (http://java.sun.com/java/products/javacard).

[33] TC39 committee, *ECMA-262 Edition 3: ECMAScript Language Specification*, ECMA standards organization (pub.), 18 January 2000.

[34] Thekkath, C. A., M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” ASPLOS IX, November 2000.

[35] Vasse, G., “IBM Extends Enhanced Data Security to Consumer Electronics Products,” IBM News Release April, 2004 (http://www.ibm.com/news/nl/nl/2006/04/nl_nl_news_20060410.html).

[36] 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.

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

[38] Yee, B., *Using Secure Co-Processors*, Ph.D. Thesis, Carnegie-Mellon Computer Science Technical Report CMU-CS-94-149, 1994.