# Practical Data Location Obfuscation

Bertrand Anckaert*, Mariusz H. Jakubowski‡,
Ramarathnam Venkatesan‡ and Chit Wei (Nick) Saw‡

*Department of Electronics and Information Systems
Ghent University
B-9000 Ghent, Belgium
banckaer@elis.UGent.be

‡Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
{mariuszj, venkie, chitsaw}@microsoft.com

January 2009
Technical Report
MSR-TR-2009-3

Software running on an open architecture, such as the PC, is vulnerable to inspection and modification. This is a concern, as software may consist of or provide access to valuable information. As a result, several defenses against program understanding and modification have been proposed in literature. The approach discussed in this paper complements existing work and focuses on hiding the actual location of data throughout the execution of the program. To achieve this, we combine three techniques: (i) periodic reordering of the heap, (ii) migrating local variables from the stack to the heap and (iii) pointer scrambling. The techniques serve to complicate static data flow analysis and dynamic data tracking. Our prototype implementation compiles C programs into a binary for which every access to the heap is redirected through a memory management unit. In order to protect traditionally stack-based variables as well, a mechanism is provided to migrate them to the heap and to adapt all accesses to those variables. Finally, an option is provided to enable pointer scrambling. If this is turned on, the program can no longer operate directly on the pointers; therefore, pointer arithmetic is intercepted as well. Experimental evaluation on benchmarks from the SPEC CPU2006 benchmark suite illustrates the type of trade-off that needs to be made for this type of protection. Balance must be struck between comprehensive protection and cost in terms of execution time and (to a lesser extent) static and dynamic memory footprint.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

# 1 Introduction

Software running on an untrusted host is inherently vulnerable to inspection and modification. Recent advances on the theoretical level have shown both negative [1] and positive [12] results on the possibility of protecting software within this severe threat model. However, little is known about the type of application one usually wants to protect.

Intuitively, any protection scheme other than a physical one depends on the operation of a finite state machine. Ultimately, given physical access, any finite state machine can be examined and modified at will, given enough time and effort [5]. However, we can observe increasing deployment of software on open architectures in scenarios where the software contains secret information or where integrity of the software is required by the business model.

For example, access to copyrighted music and video is increasingly controlled by software-based digital containers. The experience of multi-player games relies heavily on users not being able to gain competitive advantages through cheating. Software licenses may be enforced through technical protection mechanisms embedded in the software. As a final example, ad-supported software relies on the ads being correctly displayed and reported.

The above examples illustrate current demand for practical defense mechanisms. Even if theoretically secure protection is impossible, the question is more whether we can increase the time and effort required to attack software to make the benefits outweigh the costs. As an example, consider a simple "Hello World" program. Assume we want to protect this against modification of the "Hello World" message. Without countermeasures, an attacker could easily replace the characters of "World" by, e.g., "Alice" in the binary, using tools such as `strings` and hex editors. However, a limited amount of obfuscation would easily foil this straightforward attack and make it more economically viable for Alice to write her own "Hello Alice" program from scratch than to modify the existing "Hello World" program. From an economical point of view, the binary would then be sufficiently protected against this type of attack.

Practical obfuscation techniques are thus about raising the bar to a level that supersedes the incentive of the attacker. This incentive is composed of many factors, such as the perceived consequences, facilitating conditions and habits [11]. In this paper, we present a novel technique to raise the bar by hiding the location of data. This technique is orthogonal to many other software protection techniques from the domain of obfuscation and tamper-resistance. We advocate having an arsenal of software protection techniques, as we believe that combined and iterated application of different techniques can create complexity, including emergent properties due to interaction among various transformations.

At the core of our technique is a software-based Memory Management Unit (MMU). An input program is rewritten so that every operation on the heap goes through the MMU. As the MMU mediates every access to the heap, the MMU is the only component that needs to know the exact location of the data. As a result, the MMU can now periodically reorder the heap. This will make it harder for the attacker to track data when the program is running. After all,

if software protection is a cat-and-mouse game, we don't want the data to be a sitting duck.

This technique is reminiscent of the oblivious RAMs discussed by Goldreich and Ostrovsky [9]. They rightfully argue that a physically shielded CPU combined with an encrypted program is insufficient to protect the software fully, since addresses of accessed memory are not hidden; as a result, information such as loop structure may leak. They introduce the concept of oblivious RAMs to hide the original memory-access pattern. Essentially, each fetch/store cycle is replaced by many fetch/store cycles. They show how to do an on-line simulation of an arbitrary RAM by a probabilistic oblivious RAM with a polylogarithmic slowdown in the running time.

This illustrates the high cost of software protection even in the presence of a trusted hardware component. Our technique is similar, but results in a more practical compromise amongst cost, security and viability. Despite the promise of a more widespread distribution of Trusted Platform Modules (TPMs), we believe that many scenarios will not be able to rely on hardware in the years to come. These scenarios may require the correct operation of the software on legacy systems or on systems whose owners are reluctant to enable the TPM because of fears such as privacy breaches and user lock-in. Furthermore, as discussed above, less stringent security requirements may be sufficient in practical settings. Therefore, our design allows for parameterization of the extent and frequency of reordering.

As reordering the heap does not protect all data (e.g., stack-based local variables), we have added the option to migrate the local variables automatically to the heap. This is done by (i) allocating the required memory when the variable is declared, (ii) rewriting all accesses to the variable to go through the pointer, and (iii) freeing the memory when the variable goes out of scope.

Finally, we have included the option of scrambling the pointers seen by the program. If the pointers are scrambled, the program can no longer perform pointer arithmetic on those pointers, so in this case, we need to intercept pointer arithmetic as well.

The remainder of this paper is structured as follows. A more detailed description of the operation of the MMU is given in Section 2. Section 3 provides an analysis of the added complexity from the viewpoint of the attacker. More practical implications of the technique are discussed in Section 4. An experimental evaluation is given in Section 5. Related work is the topic of Section 6 and suggestions for future work are made in Section 7. Finally, conclusions are drawn in Section 8.

# 2 Operation of the MMU

The main idea behind our approach is to reorder the data on the heap periodically while retaining the original functionality of the program. The MMU will mediate all accesses to the heap, making the reordering as transparent as possible to the original program. As accessing the heap is a common operation,

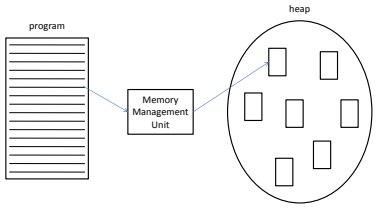


Figure 1: High-level overview

the overhead cannot be too large. Therefore, we use a mechanism derived from paging to keep track of the current location of the data.

The heap is divided into pages of a certain size $s$ (e.g., $s = 4\text{KiB}$) and we keep track of the location of each allocated page through a mapping. For every access to the heap, we randomly permute $n + 1$ pages with probability $1/p$, where $n$ and $p$ are security parameters that can be tweaked to enable a trade-off between performance and security. Additionally, the pages are encrypted. To make it harder to detect the permutation through binary similarities, we use the technique of salting. This means that before every encryption, the pages are padded with a different value, which will result in different encrypted pages, even for identical content (because of the unique salt).

Because of this reordering mechanism, the correspondence between pointers in the program's data space and the actual location of the data in memory varies over time. The correct mapping is considered to be known only to the MMU. As with traditional paging mechanisms, the tables to map pointers to the actual locations of the pages can be stored in pages as well. With 4KiB pages, for example, two levels of indirection would suffice to retrieve the requested page.

Software-based protection in the malicious-host model suffers from the absence of a nucleus of trust. In order to make claims about the security of the applied technique, we will assume that the MMU can be trusted. This is an engineering assumption. However, if we can make the trusted component small enough, the problem of protecting a generic program is reduced to the problem of protecting a very specific, smaller piece of code. The MMU can then in practice be protected by existing techniques from the domain of obfuscation and individualization. The security claims in the next section assume that the MMU can perform encryption and decryption, has access to a pseudo-random number generator, and has the memory required to swap two pages. The operation of the MMU is depicted in Figure 1.

# 3 Security Analysis

Each time a data item on a given page is read or written by the program, $n$ additional pages are pulled into the MMU with probability $1/p$. The requested operations are performed, the pages are salted differently, re-encrypted, and put back into memory in a random permutation. This is the root of confusion for the attacker.

With every step, different candidate locations exist for the location of a particular piece of data. If we assume that the accesses to the heap are random, then the following construction may be used to get the average number of possible candidates, assuming that $p = 1$.

Let's assume that there are $N$ pages in total. We tackle this problem as follows: When a page is accessed, it will be pulled into the MMU, along with $n$ additional pages. After the first timestep $t_1$, $n$ pages are written back to memory in a random permutation over the $n+1$ pages, while one page is retained in the MMU. After the first timestep, there are $n+1$ candidate locations for the page that was actually requested. We denote this as $C(t_1) = n + 1$, or the confusion at timestep $t_1$ is $n + 1$.

We therefore mark these pages "red". In the next step, $n$ random pages are taken from the $N - 1$ pages not yet in the MMU. Every taken page which was not yet marked "red" now becomes "red."

To assess the additional number of pages marked at each timestep, we note that a sequence of draws from a finite population without replacement is defined by a hypergeometric distribution. As such, the average number of marked pages in a sequence of $n$ draws from the $N-1$ candidates, of which $C(t_i)-1$ are marked at timestep $t_{i+1}$, is given by $n(C(t_i) - 1)/(N - 1)$. The average number of non-marked pages is then $n - n(C(t_i) - 1)/(N - 1)$. As a result, we obtain the following equations for the confusion:

$$\begin{aligned} C(t_1) &= 1 + n \\ C(t_{i+1}) &= C(t_i) + n - n(C(t_i) - 1)/(N - 1) \end{aligned}$$

One can easily verify that this converges to $N$, since no more candidates are added once $C(t_i)$ equals $N$. Thus, for $n = N - 1$ and $p = 1$, we would obtain true oblivious RAMs [9] under the specified assumptions.

# 4 Prototype Implementation

In this section, we will elaborate on the more practical aspects of the suggested transformation. Our prototype implementation compiles C programs into programs with the described properties. The prototype is built on top of the Phoenix framework [13], which is discussed in Section 4.1. The core of the transformation, namely redirecting all heap accesses through the MMU, will be discussed in Section 4.2. The two optional transformations — migrating local variables to the heap and scrambling the pointers in the program's data space

— will be discussed in Section 4.3 and 4.4. The next two sections (4.5 and 4.6) elaborate on the complications that arise when pointer usage crosses the boundary between code within the reach of our transformation and external code. The final section (4.7) discusses the robustness of the prototype implementation.

## 4.1 Phoenix

Our prototype implementation is built on top of Phoenix[1], a framework for building compilers and tools for program analysis, optimization and testing. Phoenix consists a comprehensive set of modules and can process different languages for different target platforms. Our prototype implementation targets the C language compiled to x86 code.

Conceptually, the prototype is a C2 plugin, which means that we graft ourselves onto the backend of the compiler. The backend reads the C Intermediate Language (CIL) generated by the front end and lowers it to machine code.

## 4.2 Mediating Heap Accesses

The MMU will be mediating every access to the heap. It will take care of the allocation and release of dynamic memory as well. Therefore, all calls to memory-management functions (malloc, calloc, realloc and free) are redirected. The current dynamic memory allocation mechanism uses buddy blocks for allocations smaller than or equal to the page size $s$. When no previously allocated memory is available for reuse, memory is allocated per page. If the block is more than twice as large as desired, it is broken in two. One of the halves is selected, and the process repeats until the block is the smallest power of two larger than or equal to the request.

A list of free blocks of different sizes is maintained. When blocks are freed, the buddy of that block is checked; if it is free as well, they can be merged once again. In the presence of free blocks, a request for memory is started from the smallest free block that can serve the request. For blocks larger than the page size $s$, we allocate the smallest number of pages sufficient to serve the request.

The pointers returned to the program do not need to correspond directly to the actual location of the data. However, if we want to allow the program to perform pointer arithmetic independently, care needs to be taken that the returned pointers can be the result of a regular allocation mechanism. This means that memory requests cannot return pointers that fall within previously allocated memory areas in the program's data space. Otherwise, during a subsequent memory access, we will not be able to determine whether the pointer was computed through an offset from the earlier allocation, or originates from the new request.

Currently, this is resolved by maintaining a straightforward relation between the pointers returned by the regular memory requests performed by our MMU behind the screens and the pointers returned to the program. The relation is not one-to-one to facilitate future memory accesses.

[1] http://research.microsoft.com/Phoenix/

As the pointers seen by the program do not correspond to the location of the data in memory, we need to intercept every read and write operation to the heap. These memory accesses may be anywhere in the memory area allocated by the program, and we need an easy way to translate the addresses to the correct location of the data at the time the request is made. Therefore, we make sure that the pointers returned to the program from a memory request are page-size aligned. On subsequent accesses, we can then easily determine the offset on the page, and a mapping will translate the page addresses as seen by program to the actual current location of the page.

## 4.3 Migrating Local Variables to the Heap

The technique discussed above only affects data on the heap. In many scenarios, the local variables of a program may contain critical information or their integrity may be crucial for uncompromised execution of the program. Therefore, we have added the option of migrating local variables from the stack to the heap.

In practice, definitions of the local variables are replaced by pointer creations through memory requests. Subsequent uses of the local variable are adapted to go through the pointer. If the variable goes out of scope, the memory is freed.

## 4.4 Pointer Scrambling

If we wish to blur the relation between the different pointers returned to program, we need to scramble them. If left unscrambled, the relation between different pointers can reveal, for example, that different smaller memory areas have been allocated on the same page, or that two independently computed pointers are related. If we want the program to be able to perform pointer arithmetic, the transformation needs to be isomorphic with respect to addition, subtraction, and the order relation.

Transforming the page addresses to page-size aligned addresses to facilitate subsequent data accesses, as described in section 4.2, has this isomorphic property. However, this restriction makes it hard to hide the relation between different pointers well. On the other hand, if the pointers are scrambled more thoroughly, the program can no longer operate on the pointers directly. As a result, if extensive pointer scrambling is turned on, we will redirect all pointer arithmetic to the MMU as well.

## 4.5 Escaping Pointers

As with any form of data obfuscation, interfacing with external code poses a challenge. The problem is that the external code is not under our control and can therefore not be modified to take the data obfuscation into account. As a result, the data needs to be put into its original format before it is passed to external code. This poses a potential security risk, as it forces the code to contain functionality to undo the transformations. Therefore, we advocate that

the reliance on external code be reduced to the absolute minimum. If possible, library functions should be internalized and included in the transformation as much as possible. Ideally, the only time data is normalized is just before I/O, as this cannot be avoided when preserving the relevant behavior of the program.

In our case, as the pointers in the program's data space no longer point to the actual data, a problem arises when pointers are passed to code that is beyond our control. This problem occurs when pointers escape to library functions or system calls. Other than internalizing external code, we discuss two main strategies to deal with this.

The first is to normalize the data. This would mean that we reorder all of the memory accessible through the escaping pointers so that the memory is in the correct layout and to pass the correct pointer to this memory area. This may require extensive normalization of the memory via a recursive process, since external code may receive aggregate arguments that in turn contain pointers to various memory locations from which other memory locations may become accessible.

The second approach is to redirect calls to those library functions to internal code that will emulate the library functions in such a way that they take the shuffled layout into account.

The first approach is more general and requires less domain-specific knowledge. However, it increases the attack surface of the MMU, as it should now contain functionality to turn the memory back into a normal layout. We would clearly like to avoid this. Therefore, we may opt instead not to include any data that may escape to library functions or system calls in the transformation. This makes sense, as this data may be revealed anyway when crossing the boundary. On the other hand, it may not always be easy to determine conservatively and accurately the data that could potentially escape during an execution of the program. These motivations, along with the fact that we are currently targeting C programs with only limited reliance on library code, have led us to switch to the second approach.

Our experimental evaluation shows that the number of library functions called from the inspected benchmarks (C programs of the SPECCPU 2006 benchmark suite) is limited. These library calls consist mainly of three types of operations: (i) memory operations such as memset, memcpy and memmove; (ii) string operations such as strlen, strcat and strcpy; and (iii) basic file operations such as read, open, and remove. The workload of creating the required function wrappers which take the shuffled memory layout into account proved to be limited. The wrappers typically divide the operation into a sequence of calls to the library function, so that the library function can operate on contiguous chunks of data (typically of size $s$). Furthermore, the SPEC benchmarks do not pass complicated structures containing pointers to the outside world.

## 4.6 Incoming Pointers

A related problem is pointers coming in from library functions. These may pose a problem because it may be hard to distinguish them from scrambled pointers.

Again, we see two possible solutions. The first is to take the "all or nothing" approach. In this solution, as soon new pointers are created (by library calls or dereferences), we will absorb them into our scheme by notifying the MMU of the creation, and using a modified pointer subsequently. This way, we are certain that the pointers have been masked (if masking is turned on), and that we need to go through the translation mechanism for every heap operation.

Once again, this may create problems if pointers to pointers or structs containing pointers are passed to the program. From our experiments, we have learned that most pointers created by external functions are the result of memory allocation, which is intercepted anyway. One notable exception are command-line arguments, which are passed as an array of strings (char pointers). As the number of arguments is known and the structure of the array is well known, we can absorb these pointers as well.

The second approach is to mark the pointers under the control of the MMU. We do this by relying on the fact that on our target operating system (Windows), the upper 2GiB of the virtual address space are reserved for the kernel. This means that the program shouldn't see pointers for which the highest bit is set. Our marking thus consists of setting this highest bit, which will identify pointers that are part of our scheme. Furthermore, this marking does not break pointer arithmetic, which means that such arithmetic can still be done directly by the program if extensive pointer scrambling is not turned on.

We have implemented both approaches and made the first approach optional.

## 4.7 Robustness of the Prototype

The transformation as implemented by our prototype requires a correct interpretation of the operation of the C code. This is no problem for the evaluated benchmarks, and the interpretation will generally be correct for programs that have not been made hard to analyze deliberately. For completeness, we report a number of limitations that may become apparent with programs that unintentionally or deliberately contain "unclean" code. In practice, the programs that one wants to protect should adhere to rigorous coding practices, as these are required for maintainability and portability.

The transformation redirects a number of operations, including function calls. This means that calls to those functions (e.g., malloc) need to be detected. Currently, our tool supports indirect function calls, but only as long as they do not escape pointers to the outside world. This is an implementation issue and could be resolved through run-time checks at the cost of additional slowdown.

Another more fundamental limitation would arise if the program performs illegitimate pointer arithmetic on scrambled pointers (e.g., by casting the pointers to integers and operating on the integers). Aside from being a portability issue, this would prevent the framework from detecting and redirecting the pointer operations.

In general, any disguised pointer usage will potentially compromise the correct operation of our transformation. As already mentioned, this does not occur

in the benchmarks from the SPEC CPU2006 benchmark suite, which have been tested extensively by others and ported across 32-bit and 64-bit systems.

# 5 Experimental Evaluation

We tested the transformation on five C benchmarks from the SPEC CPU2006 suite listed in Table 1. The obfuscated benchmarks were run and timed on a Pentium 3.0 GHz workstation with 2 GB of RAM. For timing comparisons with the unmodified benchmarks, we applied the obfuscation by adding the following individual transformations in stages. The implementation of these transformations is described elsewhere in this paper:

- Mediated Heap Access
- Migrating Local Variables to the Heap
- Pointer Scrambling

## 5.1 Benchmarks

Table 1 lists the benchmarks against which we tested.

| Benchmark | Description |
|---|---|
| 401.bzip2 | Compression |
| 429.mcf | Combinatorial optimization |
| 433.milc | Physics: quantum chromodynamics |
| 458.sjeng | Artificial intelligence: chess |
| 470.lbm | Fluid dynamics |

Table 1: Description of the Benchmarks

We ran each of the baseline benchmarks on the test machine to get the base execution timings. We then applied the obfuscation in stages as described and collected the timings of each test run. These timings were then expressed as a factor of the baseline timing to represent the overhead of the applied transformation. These results are shown in Table 2.

| Transformation | 401.bzip2 | 429.mcf | 433.milc | 458.sjeng | 470.lbm |
|---|---|---|---|---|---|
| Mediated Heap | 117.7 | 47.9 | 15.5 | 4.5 | 8.7 |
| Mediated Heap + Pointer Scrambling | 151.1 | 61.0 | 21.7 | 93.9 | 8.8 |
| Mediated Heap + Migrated Local Variables | 518.3 | 91.2 | 114.6 | 542.1 | 23.5 |
| Mediated Heap + Migrated Variables + Scrambling | 523.6 | 102.3 | 117.4 | 619.4 | 23.7 |

Table 2: Transformation Overhead for Selected Benchmarks

It should be noted that in running these tests, the transformations were applied over the entire program in order to achieve an unbiased and consistent set of results for comparison of the various techniques. As every memory access in the code is redirected through the MMU, the impact on the run times can be significant.

We also performed the tests on two simple functions that we created that repeatedly perform a specific operation in a loop to see the effect the transformations have on these particular types of operations. The first function, pseudorand, computes the function $x_n = (ax_{n-1} + b) \bmod c$ 30,000,000 times. In the program, a single variable $x$ is allocated on the heap. The second function, sumlist, calculates the summation of 30,000,000 integers stored in a linked list. In this case, all 30,000,000 list elements are allocated on the heap. The results for these tests are shown in Table 3. The difference in the overhead illustrates that the method is sensitive to the amount and type of data on which it is applied, and judicious, targeted application can significantly improve performance. Section 7 mentions some further practical ways to alleviate the slowdown.

## 5.2 Performance Issues

Even with the slowdown levels incurred by our preliminary implementation, selective application of our techniques may find some practical use. For example, applications such as DRM and access control involve Boolean checks executed outside performance-sensitive program paths. Such security tests may be required just once, a few times, or infrequently during runtime. These operations often can be made orders of magnitude slower without perceptibly affecting user experience.

When better optimized, our implementation may also be useful in more performance-oriented applications, such as stream decryption with sensitive keys. Runtime profiling and user input can help to determine which parts of the application would benefit most from our techniques. To avoid attracting attention to security-critical code, we would typically protect various unrelated parts of the application as well.

| Transformation | pseudorand | sumlist |
|---|---|---|
| Mediated Heap | 3.7 | 156.7 |
| Mediated Heap + Pointer Scrambling | 4.0 | 160.5 |
| Mediated Heap + Migrated Local Variables | 9.1 | 315.3 |
| Mediated Heap + Migrated Variables + Scrambling | 10.2 | 338.2 |

Table 3: Transformation Overhead for Simple Functions

# 6 Related Work

The technique described can be used for both obfuscation and tamper-resistance. Most existing techniques from the domain of tamper-resistance focus on protecting the integrity of code and are based on checksumming segments of the code [2, 10]. A generic attack against such schemes has been devised for the x86 through the manipulation of processor-level segments, and for the UltraSparc through a special translation look-aside buffer load mechanism [15]. Related techniques [3] hash the execution of a piece of code, while others have looked at the reaction mechanism in more detail. Once tampering is detected, appropriate action needs to be taken. If the manifestation of this action is too obvious, it can be easily tracked down. Delayed and controlled failures [8] are a way to make it harder to locate the reaction mechanism.

Software obfuscation [6, 1] aims to make programs harder to understand. There is a considerable body of work on code obfuscation that focuses on making it harder for an attacker to decompile a program and extract high level semantic information. Our technique is complementary to most existing work and focuses specifically on making it harder to detect dynamic data flow.

White-box cryptography [4] can be seen as a specific, clearly defined problem in obfuscation. Here the goal is to hide a secret key in a cryptographic software implementation in the malicious-host model. Our solution can help to defend against certain attacks based on analyzing dataflow, but should be viewed as just one component of a comprehensive software-protection toolbox.

# 7 Future Work

The prototype implementation could be improved and extended in a number of ways. Future work includes optimizing the code, especially that of the MMU to reduce the cost of the applied transformation. We continue with a discussion of a number of possible extensions.

### 7.0.1 Alternative Heap Management

The current heap management is based upon a paging mechanism, and pages are permuted periodically. Alternative schemes are possible as well. One suggested alternative approach is to keep the data in a self-balancing binary search tree. This way, data would be retrieved by looking for it in a binary tree. As the tree is self-balancing, the data reordering would be automatic. A splay-tree implementation, as suggested by Varadarajan et al. [14], could furthermore exploit data locality, as recently accessed items will be near the top of the tree.

### 7.0.2 Code Reordering

A more elaborate extension would consist of including the code (not just the data) in the reordering mechanism. This would typically be done after the

assembly representation has been generated, possibly as a link-time transformation. The code can be divided into chunks that can fit on a single page as, at this point, the exact code size is known. We could then redirect control-transfer instructions to code on other pages to go to the MMU. The MMU could then make the requested code available for execution by the CPU. Requests for code pages can be handled similarly to data accesses to permute the code and data.

#### 7.0.3 User-directed Application

The operation of our current prototype is fully automated. Furthermore, the results reported in the evaluation section are obtained by applying the transformation to the entire program. The slowdown could easily be alleviated through profiling, as a result of the 10/90 rule of thumb: 10 percent of the code is responsible for 90 percent of the execution time. If we could avoid frequently executed code, the slowdown would be considerably smaller. We believe that the reported timing information enables the reader to get a good feel of the real cost of the technique.

In practice, however, it is often the case that the programmer has domain-specific information as to which data needs to be protected and which data is less vital. Therefore, the option may be provided to mark that data at the source level and to limit the transformation to this data.

## 8 Conclusion

This paper has presented a practical approach to hiding data-access patterns in real-life programs. The technique involves shuffling and encrypting data in memory, as well as protecting pointer references. Realizing some benefits of oblivious memory accesses [9, 14], the approach complicates attacks based on dataflow analysis and memory traces. Our tool implementation may be useful standalone against certain targeted dataflow attacks, but should also be considered as an element of more comprehensive protection systems [7].

## References


[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st IACR Crypto Conference*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18, 2001.

[2] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer-Verlag, 2002.

[3] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. In

*Proceedings of the 5th Information Hiding Conference*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414. Springer-Verlag, 2002.

[4] S. Chow, P. Eisen, H. Johnson, and P. Van Oorschot. White-box cryptography and an AES implementation. In *Proceedings of the 9th Workshop on Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer-Verlag, 2003.

[5] F. Cohen. Operating system evolution through program evolution. *Computers and Security*, 12(6):565–584, 1993.

[6] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 6th International Conference on Computer Languages*, pages 28–38. IEEE Computer Society Press, 1998.

[7] N. Dedic, M. H. Jakubowski, and R. Venkatesan. A graph game model for software tamper protection. In *2007 Information Hiding Workshop*, 2007.

[8] T. Gang, C. Yuqun, and M. Jakubowski. Delayed and controlled failures in tamper-resistant systems. In *The 8th Information Hiding Conference*, 2006.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[10] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer-Verlag, 2002.

[11] M. Limayem, M. Khalifa, and W. Chin. Factors motivating software piracy: A longitudinal study. In *20th Int'l Conf. Information Systems*, pages 124–131, 1999.

[12] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *Eurocrypt*, 2004.

[13] Microsoft Corporation. Phoenix compiler framework, 2008.

[14] A. Varadarajan and R. Venkatesan. Limited obliviousness for data structures and efficient execution of programs. Technical report, Microsoft Research, 2006.

[15] G. Wurster, P. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *The 26th IEEE Symposium on Security and Privacy*, pages 127–138, 2005.