Architecture Support for 3D Obfuscation

Mahadevan Gomathisankaran and Akhilesh Tyagi, *Member*, *IEEE*

Abstract—Software obfuscation is defined as a transformation of a program $\mathcal{P}$ into $\mathcal{T}(\mathcal{P})$ such that the whitebox and blackbox behaviors of $\mathcal{T}(\mathcal{P})$ are computationally indistinguishable. However, robust obfuscation is impossible to achieve with the existing software only solutions. This results from the power of the adversary model in Digital Rights Management systems, which is significantly more than in the traditional security scenarios. The adversary has complete control of the computing node—supervisory privileges along with the full physical as well as architectural object observational capabilities. In essence, this makes the operating system (or any other layer around the architecture) untrustworthy. Thus, the trust has to be provided by the underlying architecture. In this paper, we develop an architecture to support 3D obfuscation through the use of well-known cryptographic methods and show how it provides copy-protection, IP-protection, and tamper-resistance.

Index Terms—Obfuscation, digital rights management, secure systems architecture.

1 Introduction

Digital rights management (DRM) deals with intellectual property (IP) protection and unauthorized copy protection. Software piracy alone accounted for $13 billion annual loss [1] to the software industry in 2002. Software digital rights management traditionally consists of watermarking, obfuscation, and tamper-resistance. All of these tasks are made difficult due to the power of adversary. The traditional security techniques assume the threat to be external. The system itself is not an adversary. This provides a *safe haven* or *sanctuary* for many security solutions. However, in DRM domain, the OS itself is not trustworthy. On the contrary, OS constitutes the primary and formidable adversary.

Any software-only solution to achieve DRM seems to be inadequate. In the end, in most scenarios, it reduces to the problem of *last mile* wherein only if some small kernel of values could be isolated from the OS (as an axiom), the entire schema can be shown to work. At this point, it is worth noting that even in Microsoft's next generation secure computing base (NGSCB) [2], the process isolation from OS under a less severe adversary model is performed with hardware help. The NGSCB's goal is to protect the process from the OS corrupted by external attacks by maintaining a parallel OS look-alike called *nexus*. The *nexus*, in turn, relies upon a hardware Security Support Component (SSC) for performing cryptographic operations and for securely storing cryptographic keys.

The trusted computing group consisting of AMD, HP, IBM, and Intel, among many others, is expected to release Trusted Platform Module (TPM) [3], to provide the SSC. The TPM is designed to provide such a root of trust for storage, for measurement, and for reporting. Hence, we believe that TPM provides building blocks for the proposed architecture. However, we identify additional capabilities needed to support robust 3D obfuscation. The proposed architecture obfuscation blocks can absorb TPM functionality (based on the released TPM 1.2 specifications [4]).

This paper is organized as follows: Section 2 describes the obfuscation problem and its interaction with the existing cryptographic solutions. Section 3 discusses earlier proposed research and their drawbacks. Section 4 explains the basic building blocks of *Arc3D* and provides a high-level overview. Section 5 provides operational details of *Arc3D* system. We describe various attack scenarios in Section 6. Section 7 gives the performance analysis of *Arc3D*. Section 8 concludes the paper.

2 The Problem

The attributes that need to be supported by a DRM system are as follows:

1. Associability of Software to a particular CPU. (*copy-protection*),
2. Verifiability of the CPU's authenticity/identity. (*copy-protection, IP-protection*),
3. Binary file, conforming to a standardized structure, should not reveal any IP of the software through static analysis-based reverse engineering. (*IP-protection—static obfuscation*),
4. Any modification of the binary file should make the software unusable. (*IP-protection—tamper-resistance*), and
5. The program execution parameters visible outside CPU should not reveal any IP of the software. (*IP-protection—dynamic obfuscation*)

The first two problems are analogous to the real-life problem of establishing trust between two parties followed by secret sharing on a secure encrypted channel. This is a well-analyzed problem and solutions like Pretty Good Privacy (PGP) exist which uses a trusted Certification Authority (CA). This approach has been used in almost all the earlier research dealing with copy-protection ([5], [6]) and we too will use a similar approach.

The third problem requires prevention (minimization) of information leak from the static binary file/image. This could

• *The authors are with the Department of Electrical and Computer Engineering, Iowa State University, 1401 Coover Hall, Ames, IA 50011. E-mail: {gmdev, tyagi}@iastate.edu.*

Manuscript received 20 May 2005; revised 30 Nov. 2005; accepted 7 Dec. 2005; published online 22 Mar. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0167-0505.

be viewed as the problem of protecting a message in an untrustworthy channel. One possible solution is to encrypt the binary file (the solution adopted by XOM [6] and ABYSS [5]). An alternative approach would recognize that the binary file is a sequence of instructions and data, with an underlying structure. Static obfuscation ([7], [8]) attempts to exploit a smaller subset of these program level structure attributes.

The fourth problem requires the binary image to be tamper resistant. Any modifications to the binary image should be detectable by the hardware. Message Digest, which is a one-way hash of the message, could be used to solve this problem. This once again is a generic solution which is applicable to any message transaction that does not use any specific properties of a binary image. We rely upon obfuscation to provide the tamper-resistance in the following way: Tampering gains an advantage for the adversary only if the properties of the tampering point—the specific instruction or data at that point—are known. However, obfuscation prevents the adversary from associating program points with specific desirable properties (such as all the points that have a branch, call sites, to a specific procedure, or all the data values that point to a specific address). Hence, most tampering points are randomly derived, resulting in the disabling of the program, which we do not consider to be an advantage to the adversary in the DRM model where the adversary/end user has already purchased rights to disable the program.

The fifth problem dictates that the CPU not trust anything outside its own trusted perimeter including any software layer. The problem is simplified by the fact that CPU can halt its operations once it detects any untrustworthy behavior. The attributes of the application program execution trace space, which the CPU has to protect, can be thought of as having three dimensions, namely, instructions (content), addresses at which the instructions are stored (address sequencing), and the temporal sequence of accesses of these addresses (second-order address sequencing). All of these three dimensions have to be protected in order to prevent any information leakage. This holds true even for data.

3 Previous Research

ABYSS [5] was proposed as a software protection architecture. It supports both *protected* and *unprotected* processes. Protected processes are executed in a *protected processor* and are encrypted. Drawbacks of ABYSS include nonscalability and unexplained OS interactions.

XOM [6] is a compartmentalized—a process in one compartment cannot access data from another compartment—machine. Application is encrypted with a symmetric key which, in turn, is encrypted with public asymmetric key of XOM. HIDE [9] is an extension of XOM. It points out the fact that XOM does not protect the time order of the address trace. Hence, even if the instructions (and data) themselves are encrypted, the address trace gives the adversary power to deduce the control flow graph (CFG).

Goldreich and Ostrovsky [10] offered one of the first schemes for software protection. They extended the oblivious Turing machine model to oblivious RAMs. An oblivious RAM presents a memory access footprint that does not depend on the program input. This prevents information leak about the program CFG. Oblivious RAM

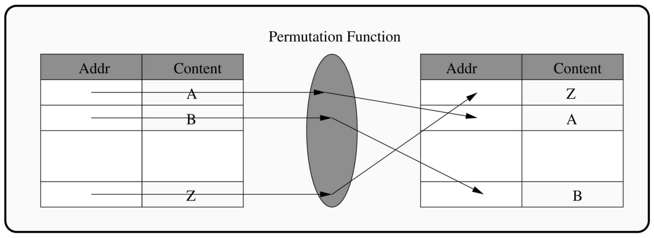

Fig. 1. Weakness of HIDE approach.

requires $O(\sqrt{m})$ steps for each original memory access. This is a very high overhead for present day architectures.

DS5002FP [11] is a secure 8-bit microcontroller from Dallas Semiconductor which uses bus-encryption. The DS5002FP implements three on-chip block-cipher functions E_A for 17-bit address-bus encryption, E_D for 8-bit data-bus encryption, and E_D^{-1} for 8-bit data-bus decryption. The encryption functions are fixed unless changed by uploading a new secret key. This allows the adversary to build up information by running the program multiple times and observing its behavior. Kuhn [12] proposed such an attack to extract the secrets stored in a DS5002FP microcontroller.

3.1 Obfuscation

We use the term *obfuscation* in a different way than the earlier literature ([13], [14]). We refer to any obfuscation mechanism that hides the control flow from static analysis through a binary image or CFG transformation as *static obfuscation*. However, these obfuscation mechanisms cannot defeat an adversary with access to architecturally visible parameters, such as a memory address bus. Our use of the term *obfuscation* refers to schemes that hide dynamic execution model parameters as well. Note that, in our obfuscation model, even the instructions themselves can be transformed into potentially invalid ones.

3.2 Analysis

Almost all of the earlier research, except HIDE, does not hide the temporal sequencing of memory accesses. Neither do these solutions exploit the software specific properties. The solution proposed by HIDE to prevent information leak through the address and memory bus is weak. This is because the adversary can see the contents of the memory before and after an address-permutation. It is possible because the encryption function applied to the contents is not address dependent. Hence, for instance, if the contents at two distinct addresses A_i and A_j are also distinct C_{A_i} and C_{A_j} then the following information leak path exists. For a program sequence within a loop, when instructions reoccur at the address and instruction buses, HIDE permutes the addresses within a page for the second (or subsequent) iteration. If A_i is permuted to a new address $\pi(A_i)$ the contents at $\pi(A_i)$ would still appear as C_{A_i}. Hence, a simple comparison would be able to determine the permutation π. Fig. 1 illustrates this fact. Thus, it takes only $\frac{N(N+1)}{2}$ comparisons to reverse-engineer the permutation, where N is the permutation size. Assuming that there are 1,024 cache-blocks in a page, the strength of such a permutation is less than 2^{20}. Even in the chunk mode, which performs these permutations in a group of pages, the complexity grows only linearly and, hence, could be easily broken.

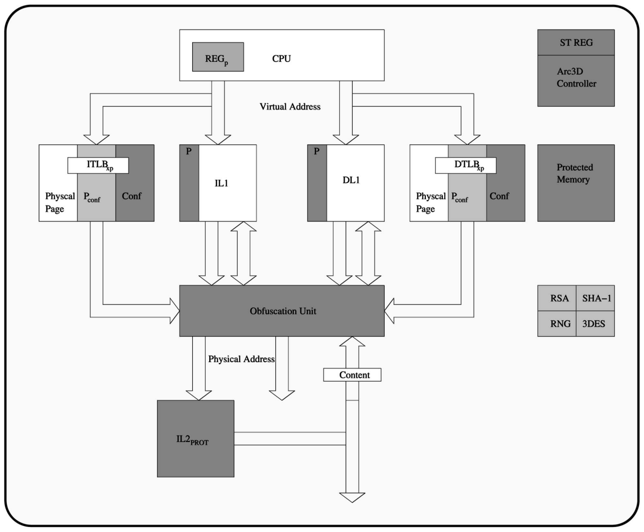

Fig. 2. Overall Schema of *Arc3D* Architecture.

The proposed architecture *Arc3D* addresses all these issues. Moreover, computational efficiency of proposed methods is a key criterion for inclusion in *Arc3D*. We make use of software structure to provide obfuscation and tamper-resistance efficiently.

4 Proposed Architecture: Arc3D

The overall *Arc3D* architecture is shown in Fig. 2. The main affected components of the microarchitecture are the ones that handle virtual-addresses. These components include the translation lookaside buffer (TLB) and page table entries (PTE). We first describe the objectives of the obfuscation schema.

4.1 Obfuscation Schema

The goal of obfuscation is to remove the correlation between

1. the CFG and the static binary image and
2. the static binary image and the dynamic execution image.

Traditional static obfuscation techniques try to obscure disassembling and decompilation stages to remove the correlation between the static image and the CFG. But these techniques are transparent to architecture and do not remove the correlation between the static image and the dynamic execution image. Thus, an adversary monitoring the address traces could very well extract the CFG.

We use *architecture aware* obfuscation of both *sequence* and *content* to achieve this goal. Static obfuscation or obfuscation of the static binary image is achieved by permuting the instruction sequence with π_S and obfuscating the instructions with C_s. Another pair of sequence permutation and content obfuscation functions which are dynamically chosen, π_D and C_D, help achieve dynamic obfuscation. These four functions, π_S, C_s, π_D, and C_D form the program secret which is guarded by the trusted component of the architecture. Fig. 3 shows these obfuscation functions in operation.

4.2 Overall Schema

As stated earlier, Fig. 2 shows the global schema for the proposed architecture. The shaded areas are the additional

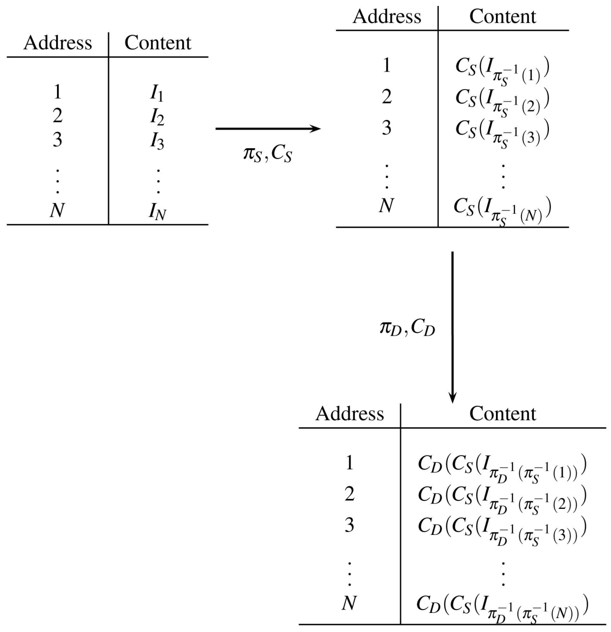
Address | Content
1 I_1
2 I_2
3 I_3
$\vdots$ $\vdots$
N I_N
π_S, C_S
Address | Content
1 $C_S(I_{\pi_S^{-1}(1)})$
2 $C_S(I_{\pi_S^{-1}(2)})$
3 $C_S(I_{\pi_S^{-1}(3)})$
$\vdots$ $\vdots$
N $C_S(I_{\pi_S^{-1}(N)})$
π_D, C_D
Address | Content
1 $C_D(C_S(I_{\pi_D^{-1}(\pi_S^{-1}(1))})$
2 $C_D(C_S(I_{\pi_D^{-1}(\pi_S^{-1}(2))})$
3 $C_D(C_S(I_{\pi_D^{-1}(\pi_S^{-1}(3))})$
$\vdots$ $\vdots$
N $C_D(C_S(I_{\pi_D^{-1}(\pi_S^{-1}(N))})$

Fig. 3. Static and Dynamic obfuscation.

components of *Arc3D* over the base architecture. Shading hues also indicate the access rights as follows: The lightly shaded areas contain information accessible to the outside world, i.e., OS. The darkly shaded areas contain secret information accessible only to *Arc3D*. *Arc3D* has two execution modes, namely, *protected* and *unprotected* mode. It has a protected register space REG_p which is accessible only to a protected process.

The core of *Arc3D* functionality is obfuscation. It is achieved by modifying the virtual-address translation path—translation look aside buffer (TLB)—of the base architecture. In addition to holding the virtual-address to physical-address mapping, page table entry (PTE), the TLB has the obfuscation configuration (P_{conf}). This P_{conf} is essentially the shared secrets C_S, C_D, π_S, π_D in encrypted form. In order to avoid frequent decryption, *Ar3D* stores them in decrypted form in $Conf$ section of TLB_{xp}. This section of TLB is updated whenever a new PTE is loaded into TLB_{xp}. *Arc3D* assumes parallel address translation paths for data and instructions and, hence, Fig. 2 shows DTLB and ITLB separately.

The address translation for a protected process occurs in the obfuscation unit. Sections 4.4 and 4.5 explain in detail the address sequence and content obfuscation algorithms, respectively. *Arc3D* uses the same logic for both static and dynamic obfuscations. The basis of these obfuscations is the permutation function which is explained in Section 4.3. *Arc3D* has a protected L2 cache, which is accessible only to a protected process, thus providing temporal order obfuscation.

The *Arc3D* controller provides the following interfaces (APIs) which enable the interactions of a protected process with the OS.

1. $start\_prot\_process$: Allocate the necessary resources and initialize a protected process.
2. $exit\_prot\_process$: Free the protected resources allocated for the current protected process.

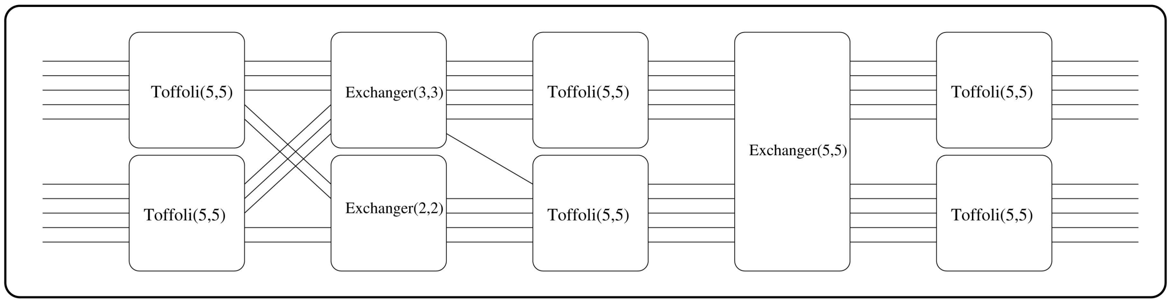

Fig. 4. Reconfigurable bijective obfuscation unit.

3. $ret\_prot\_process$: Return to the current protected process from an interrupt handler.
4. $restore\_prot\_process$: Restore a protected process after a context switch.
5. $transfer\_prot\_process$: Fork the current protected process.

These APIs and their usage are explained in detail in Section 5.

4.3 Reconfigurable Bijective Function Unit

Obfuscation unit is a major component of *Arc3D*. This unit is responsible for generating *bijection* functions π. There are $2^n!$ possible n-bit reversible functions. Reconfigurable logic is well-suited to generate a large dynamically variable subset of these reversible functions. Fig. 4 shows one such schema for permutation of 10 address bits (specifying a page consisting of 1,024 cache-blocks). Before explaining the blocks of Fig. 4, we observe that there are $\left(2^{2^n}\right)^n$ possible functions implemented in a $n \times n$ lookup table (LUT) or n n-LUTs. But, only a subset of them are bijective. We wish to implement only reversible (conservative) gates ([15], [16]) with LUTs.

Definition 1. *A Toffoli gate, $Toffoli(n,n)(C,T)$, is defined over a support set $\{x_1, x_2, \dots, x_n\}$ as follows: Let the control set $C = \{x_{i1}, x_{i2}, \dots, x_{ik}\}$ and the target set $T = \{x_j\}$ be such that $C \cap T = \emptyset$. The mapping is given by $Toffoli(n,n)(C,T)$ $[x_1, x_2, \dots, x_n] = [x_1, x_2, , x_{j-1}, z, x_{j+1}, , x_n]$, where $z = x_j \oplus (x_{i1} \wedge x_{i2} \wedge \dots \wedge x_{ik})$.*

Both Fredkin and Toffoli [15] and Toffoli [17] have defined classes of reversible gates. We use *Toffoli(5,5)* gates with 5-input bits and 5-output bits in our scheme, as shown in Fig. 4. However, we could easily replace them by *Fredkin(5,5)* gates. The domain of configurations which can be mapped to each of the LUTs consists of selections of sets T and C such that $T \cap C = \emptyset$. For a support set of five variables, the number of unique reversible Toffoli functions is $4\binom{5}{1} + 3\binom{5}{2} + 2\binom{5}{3} + \binom{5}{4}$. Each of these terms captures control sets of size 1, 2, 3, and 4, respectively. Ignoring control sets of size 1, we get a total of 55 reversible functions. Thus, total permutation space covered by all six of these gates is $(55)^6 \approx 2^{34}$. There are several redundant configurations in this space. We estimated the redundancy through statistical analysis to be less than 0.3 percent.

The exchanger blocks shown in Fig. 4 perform a *swap* operation. It has two sets of inputs and two sets of outputs. The mapping function is $S_{ok} = S_{ik}$ if $X = 0$, and $S_{ok} = S_{i\bar{k}}$ if $X = 1$, where, S_{ik} is the input set, S_{ok} is the output set, X is configuration bit, and k is 0 or 1. Since *exchange* is also bijective, the composition of *Toffoli* gates and *exchangers* leads to a bijective function with large population diversity. Other interesting routing structures may also guarantee bijections. But, a typical FPGA routing matrix configuration will require extensive analysis to determine if a given routing configuration is bijective. One point to note here is that we chose to implement a $10\ bit$ permutation function with $Toffoli(5,5)$ gates instead of a direct implementation of $Toffoli(10,10)$. This is because an n-LUT requires 2^n configuration bits and, hence, 10-LUTs are impractical in the reconfigurable computing world.

Having fixed the reconfigurable logic to perform the obfuscation (permutation), we need to develop a schema for the LUT configuration. A simple mechanism would be to store all the 55 possible configurations at each of the LUTs (similar to DPGA of DeHon [18]). In addition to $4\ input\ bits$, each LUT will also have *six configuration bits* to choose from one of the 55 configurations (assuming some configurations are repeated to fill the 64 locations), as shown in Fig. 5. In Fig. 5, $A_0A_1A_2A_3A_4$ represent the input address bits, f_0, f_1, f_2, $\dots$, f_{63} represent the 64 configurations and $C_0C_1C_2C_3C_4C_5$ represent the configuration bits. Each of the *exchanger* blocks also requires 1 configuration bit. Thus, a total of 39 configuration bits are needed by the reversible logic of Fig. 4.

4.4 Obfuscating the Sequence

We can use the reconfigurable permutation unit defined in Section 4.3 to achieve sequence obfuscation. Note, however, that even though we have shown the circuit for a 10-bit

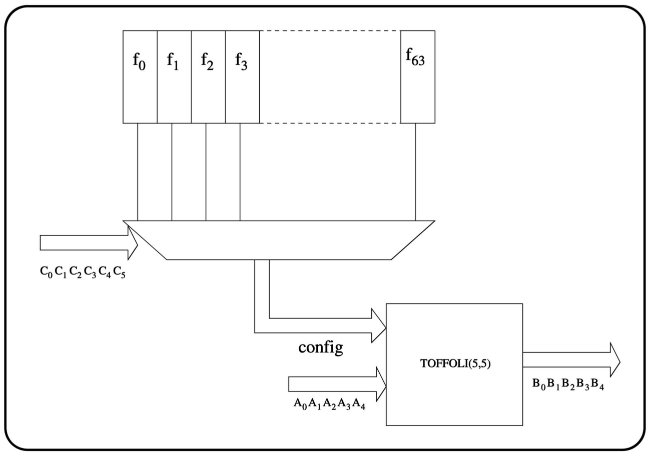

Fig. 5. Configuration selection for each LUT.

permutation, the methodology is applicable to an arbitrary number of address bits. We believe that at least 10 address bits need to be permuted in order to have a reasonably large permutation space. The choice of 10-bits is also dictated by the structure of the software. Software objects, both instruction and data, are viewed by the architecture in various granularities. The RAM memory resident objects are viewed in the units of *pages*. The cache resident objects on the other hand are viewed in the units of *blocks*. This argues for the obfuscation boundaries defined by these units. Hence, we obfuscate the sequence of *cache-blocks* within a page. Any sequence obfuscation within *page* level needs to interact with the page management module of the OS. If the obfuscated sequence crosses the *page* boundary, the permutation function (π) has to be exposed to the OS. This is the reason why we cannot obfuscate sequences of pages. In the other direction, permuting the sequences of subunits of cache-blocks seriously affects the locality of cache resulting in severe performance degradation. Moreover, since the contents of a cache-block are obfuscated, the information leak through the preserved, original sequence of cache sub-blocks is minimized. Considering a *page* size of 64 KB with 64 B *cache-blocks*, as is the case with Alpha-21264, we get 1,024 *cache-blocks* per page, i.e., 10-bits of obfuscation.

4.5 Obfuscating the Contents

In cryptography, the *one time pad* (OTP), sometimes known as the *Vernam cipher*, is a theoretically unbreakable method of encryption where the plaintext is transformed (for example, XOR) with a random *pad* of the same length as the plaintext. The structure of the software objects determines the protection granularities once again. We can consider a program as a sequence of fixed sized messages, i.e., *cache-blocks*. If we have unique OTPs for each one of the cache-blocks in the software, the contents are completely protected. However, the storage and management of that many OTPs is highly inefficient. Nonetheless, we at least have to guarantee that every *cache-block* within a *page* has a unique OTP. This is to overcome the weakness in HIDE (as explained in Section 3, Fig. 1). If the adversary-visible contents of the memory locations are changed after each permutation (as with unique cache-block OTP per page), then n-bit permutation is $2^n!$ strong. This is in contrast with the strength of the order of 2^n exhibited by the original HIDE scheme.

In order to provide a unique OTP per cache-block per page, one option is to generate a random OTP mask for each cache-block for each page. A more efficient solution, however, is to pregenerate N_b OTPs for every cache-block within a page ($OTP[b_i]$ masks for $0 \leq b_i < N_b$ for a cache with N_b blocks). However, the association of an OTP with a cache-block is randomized with the π_c function. The π_c function can be chosen differently for each page to provide us with a unique OTP per cache-block. This simplifies the hardware implementation of content obfuscation unit as well since each page is processed uniformly in this unit except for the π_c function. Hence, a program image will need to provide a page of OTPs which will be used for all its pages. It also needs to specify a unique mapping function π_c per page. Since we already have the reconfigurable permutation logic of Section 4.3 in *Arc3D*, we can use it to implement the function π_c as well. This results in 39-bits per page overhead for the specification of the content obfuscation. Note that the OTP-based content encryption can be easily replaced by any other *bijective* function.

4.6 Obfuscating Temporal Order (Second-Order Address Sequences)

The second-order address sequences are derived from iterative control constructs within a program. Consider a loop of k instructions which is iterated N times. The expected address sequence in such an execution is $\{I_{(0,0)}, I_{(1,0)}, \ldots, I_{(k-1,0)}\}$, $\{I_{(0,1)}, I_{(1,1)}, \ldots, I_{(k-1,1)}\}$, $\ldots$, $\{I_{(0,N-1)}, I_{(1,N-1)}, \ldots, I_{(k-1,N-1)}\}$, where $I_{i,j}$ denotes the $i\text{th}$ instruction in the loop body in the $j\text{th}$ loop iteration. In this sequence, if an adversary is able to tag the boundaries of loop iteration, a strong correlation exists between successive iteration traces: $\{I_{(0,l)}, I_{(1,l)}, \ldots, I_{(k-1,l)}\}$ and $\{I_{(0,l+1)}, I_{(1,l+1)}, \ldots, I_{(k-1,l+1)}\}$. In fact, instruction I_0 occurs in the same relative order from the loop sequence start point in both (or all) the iterations. This allows an adversary to incrementally build up information on the sequencing. Whatever sequence ordering is learned in iteration l is valid for all the other iterations. The second-order address sequence obfuscation strives to eliminate such correlations between the order traces from any two iterations.

Interestingly, the second-order address sequence obfuscation is an inherent property of a typical computer architecture implementation. The access pattern we observe outside the CPU is naturally obfuscated due to various factors like *caching*, *prefetching*, and several other *prediction* mechanisms aimed at improving the performance. But, these architecture features are also controllable, directly or indirectly, by the OS and other layers of software. For example, the adversary could flush the cache after every instruction execution. This renders the obfuscation effect of *cache* nonexistent. To overcome such OS directed attacks, it is sufficient to have a reasonably sized *protected-cache* in the architecture which is *privileged* (only accessible to secure processes). We expect a *cache* of the same size as a *page*, in our case 64 KB, should be able to mask the effects of loops. Encrypted or content-obfuscated *cache-blocks* already obfuscate CFGs (within the cache-block). This is because a 64 B cache-block contains 16 instructions if we assume instructions of length 32-bits.

5 Arc3D in Operation

We have developed and described all the building blocks of *Arc3D* in Section 4. In this section, we explain its operation with respect to the software interactions in detail, from software distribution to the management of a protected process by the OS using the APIs provided by *Arc3D*.

5.1 Software Distribution

Arc3D provides both *tamper-resistance* and *IP-protection* through obfuscation. Hence, a software vendor should be able to obfuscate the static image of a binary executable. Moreover, a mechanism to distribute the *static obfuscation* configuration from the vendor to *Arc3D* needs to be supported. This configuration constitutes the shared secret between the vendor and *Arc3D*. Trust has to be established between *Arc3D* and the vendor in order to share this secret. Once the trust is established, the binary image along with the relevant configuration can be transferred to *Arc3D*.

5.1.1 Trust Establishment

We assume that there exist protected elements within the CPU which are accessible only to the architecture, and not to any other entities. We also assume that every CPU has a *unique identity*, namely, its *public-private key pair* $(E_k{}^+, E_k{}^-)$.

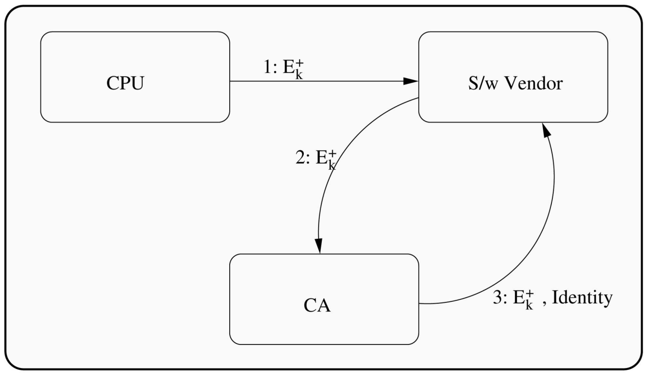

Fig. 6. Three party trust model.

This key pair is stored in the protected space of the CPU. A TPM's endorsement key pair constitutes such an identity. Public part of this key pair, $E_k{}^+$, is distributed to a *certification authority* (CA). CA verifies the CPU vendor's authenticity, associates $E_k{}^+$ with the CPU vendor's identity and other information (such as model number, part number, etc.). Any party entering a transaction with the CPU (such as a software vendor) can query the CA with $E_k{}^+$ in order to establish trust in the CPU. Since CA is a trusted entity, the data provided by CA can also be trusted. This is very similar to the PGP model of trust establishment and is shown in Fig. 6.

An important point to note here is that the trust establishment and the key management mechanisms do not constitute the crux of *Arc3D* architecture. *Arc3D* could use any model/policy for this purpose. We use this model for illustration purposes only. It could very well be adapted to use the TPM [3] model.

5.1.2 Binary Image Generation

The software vendor receives $E_k{}^+$ from the CPU. It queries the CA to derive the architecture level specifications of the CPU, relevant for static obfuscation which include details such as *cache-block* size and the minimum supported *page* size. The software vendor generates the binary file targeted at the appropriate cache-block and page sizes. It generates two sets of random configurations per page. One configuration is to obfuscate the sequence of *cache-block* addresses within a page (π_S) and the second configuration is to obfuscate the association of OTPs with *cache-block* address (π_{c_s}). The content obfuscation requires the software vendor to further generate a page sized OTP (OTP_s, $OTP_s[b_i]$ for all $0 \leq b_i < N_b$). These functions can then be used along with the FPGA obfuscation unit in a CPU or with a software simulation of its behavior to generate the obfuscated binary file.

Algorithm 1. Page Obfuscation Function: *page_obfuscate*

```
Required Functions
$F_{obf}(conf\_sel, addr) \Leftarrow$ Reconfigurable Obfuscation Unit
Inputs
$OTP_{arr} \Leftarrow$ array of OTP
$page_i \Leftarrow$ input page
$conf_{seq} \Leftarrow$ conf_sel for sequence obfuscation
$conf_{cont} \Leftarrow$ conf_sel for content obfuscation
$N_b \Leftarrow$ number of cache blocks in a page
Outputs
$page_o \Leftarrow$ output of page
Function
for $k = 0$ to $N_b - 1$ do
    $out = F_{obf}(conf_{seq}, k)$
    $l = F_{obf}(conf_{cont}, k)$
    $OTP = OTP_{arr}[l]$
    $page_o[out] = page_i[k] \oplus OTP$
end for
```

Although this kind of obfuscation is applicable to any binary image, the software vendor could enforce additional properties on the target CPUs. For instance, it can restrict the distribution only to those machines which have a certain minimum *cache-block* size and *page* size, as both these parameters affect the strength of obfuscation. A suggested minimum for these parameters is 64 B and 64 KB, respectively. The basis of static obfuscation is a page obfuscation function (*page_obfuscate*) which takes an input page, an OTP page, and configurations for both address-sequence and content obfuscation functions. It produces an obfuscated output page. The outline of this algorithm is shown in Algorithm 1. The algorithm for *static obfuscation* is shown in Algorithm 2.

Algorithm 2. Static Obfuscation Function: $stat\_obfuscate$

```
Inputs
$N_p \Leftarrow$ number of pages in the binary
$Page_{arr} \Leftarrow$ array of pages
Function
$p \Leftarrow$ temporary page
Generate random page of OTP ($OTP_s$)
for $k = 0$ to $N_p - 1$ do
  if $Page_{arr}[k]$ to be protected then
    Generate random $S_{seq}$
    Generate random $S_{cont}$
    $Page_{arr}[k].p_{conf} = K_s\{S_{seq}, S_{cont}\}, HMAC$
    $p = page\_obfuscate(S_{seq}, S_{cont}, OTP_s, Page_{arr}[k])$
    $Page_{arr}[k] = p$
  end if
end for
```

For every protected page the software vendor generates S_{seq}, the configuration for sequence obfuscation (corresponding to π_S), and S_{cont}, the configuration for content obfuscation (corresponding to π_{c_s}). It uses *page_obfuscate* to obfuscate the page and associate the configuration information with the page. This is shown in Algorithm 2. Even for pages which are not loaded, an obfuscation function could be associated. Note that *Arc3D* needs a standardized mechanism to garner these functions. This could be done by extending the standard binary format, like ELF, to hold the sections containing the configurations. The configurations have to be guarded and, hence, need to be encrypted before being stored with the binary image. The software vendor has to generate a key, K_s, specific to this installation to support such encryption. All page level configurations, S_{seq} and S_{cont}, are encrypted with this K_s. An HMAC [19] of these encrypted configurations is also generated. HMAC is a keyed hash which will allow *Arc3D* to detect any tampering of the encrypted configurations. Let P_{conf} represent the encrypted configurations and its HMAC and let S_{conf} represent the section containing P_{conf} of all the pages. The new binary format should carry encrypted configurations and its HMAC for every protected page. The page containing the cache-block OTPs also needs to be stored. This page is also encrypted with K_s. Its HMAC is computed

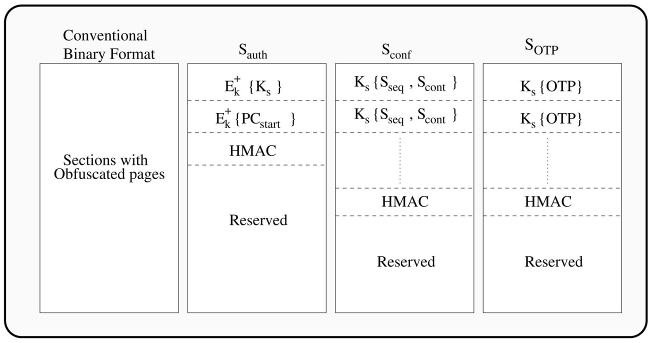

Fig. 7. Extended binary format.

as well. A new section S_{OTP} is created in the binary file and the encrypted OTP page and its HMAC are added to it.

Algorithm 3. Software Distribution
```
1: Get $E_k^+$ from CPU
2: Contact CA and validate $E_k^+$
3: Generate $K_s$
4: Generate $conf\_seq$, $conf\_cont$ for every page to be protected
5: Generate $OTP$ page
6: Do $stat\_obfuscate$
7: Generate $S_{auth}$ and add it to binary file
8: Generate $S_{conf}$ and add it to binary file
9: Generate $S_{OTP}$ and add it to binary file
10: Send the binary file to CPU
```

In order for the CPU to be able to decrypt the program, it needs the key K_s. This is achieved by encrypting K_s with E_k^+ and distributing it along with the software. Now only the CPU with the private key E_k^- can decrypt the distributed image to extract K_s. The entry point of the program also needs to be guarded. Several attacks are possible if the adversary could change the entry point. Hence, the entry point is also encrypted with K_s. Once again we need to use HMAC to detect any tampering. Hence, S_{auth}, the authorization section, consists of $E_k^+\{K_s, PC_{start}\}, HMAC$. These extended sections are shown in Fig. 7. The complete algorithm for software distribution is shown in Algorithm 3.

5.2 Management of Protected Process

We now explain the OS use of the *Arc3D* APIs to manage a protected process. We will also show how seamlessly it can be integrated with the existing systems while providing the guarantees of *tamper-resistance* and *copy-protection*.

5.2.1 Starting a Protected Process

Arc3D has two execution modes, 1) protected and 2) normal, which are enforced without necessarily requiring the OS cooperation. When the OS creates a process corresponding to a protected program, it has to read the special sections containing S_{auth} and per-page configuration P_{conf}. *Arc3D* has an extended translation lookaside buffer (TLB_{xp}) in order to load these per-page configurations. The decision whether to extend the page table entry (PTE) with these configuration is OS and architecture dependent. We consider an architecture in which the TLB misses are handled by the software. Hence, the OS can maintain these associations in a data structure different from PTEs. This will be efficient if very few protected processes (and, hence, protected pages) exist. This method is equally well applicable to a hardware managed TLB wherein all the PTEs have to follow the same structure.

The OS, before starting the process, has to update extended TLB with P_{conf}, for each protected page. Additionally, for every protected page, the OS has to set the protected mode bit P. This will be used by the architecture to decide whether to use the obfuscation function. Note that by entrusting the OS to set the P bit, we have not compromised any security. The OS does not gain any information or advantage by misrepresenting the P bit. For example, by misrepresenting a protected page as unprotected, the execution sequence will fail as both instructions and address sequences will appear to be corrupted. This is followed by the OS providing *Arc3D* with a pointer to S_{auth} and a pointer to S_{OTP}.

The OS executes *start_prot_process* to start the protected process execution. This causes *Arc3D* to transition to *protected* mode. *Arc3D* decrypts S_{auth} and checks its validity by generating its HMAC. If there is any mismatch between the computed and stored HMACs, it raises an exception and goes out of *protected* mode. If HMACs match, then *Arc3D* can start the process execution from PC_{start}. However, the address sequence generated at the address bus will expose the π_S function through one-to-one correspondence with the static binary image sequence. This compromises the static obfuscation. As explained in HIDE [9], the address sequence information suffices to reverse engineer the IP without even knowing the actual instructions. Hence, *Arc3D* performs one more level of obfuscation, called *dynamic obfuscation*, on protected pages to avoid these scenarios.

Dynamic obfuscation is very similar to the *static obfuscation*. It consists of two independent obfuscation functions per page, one to obfuscate the sequence of *cache-block* addresses and the other to obfuscate the contents of *cache-blocks*. When $start\_prot\_process$ is executed, *Arc3D* generates an OTP page (OTP_d). This OTP_d needs to be stored in memory so that it can be reloaded at a later point after a context switch. We use the section S_{OTP} to store OTP_d. *Arc3D* has sufficient internal space to hold both OTP_s and OTP_d at the same time. It reads S_{OTP} and decrypts OTP_s, validates the HMAC, and then loads it into the obfuscation engine. It then encrypts OTP_d with K_s and generates its HMAC which is appended to S_{OTP}. We assume that the space for OTP_d in the section S_{OTP} has already been allocated at compile time.

Arc3D then scans the TLB and validates P_{conf} for every protected page that has been loaded in the main memory. It then generates D_{seq} and D_{cont} configurations (corresponding to π_D and π_{c_d}) for each one of those pages and appends them to their P_{conf}. TLB_{xp} which has been extended to hold P_{conf}, also has protected space per TLB entry which only *Arc3D* can access. This space will be used by *Arc3D* to store the decrypted S_{seq}, S_{cont}, D_{seq}, D_{cont} configurations, so that decryption need not be done for every TLB access. *Arc3D* contains temporary buffer of twice the *page* size to perform the obfuscation. Hence, it reads a complete page from RAM and applies *page_obfuscation* and then stores it back in RAM. Algorithm for dynamic obfuscation is shown in Algorithm 4.

Algorithm 4. Dynamic Obfuscation Function: $dyn\_obfuscate$

```
Inputs
$N_{TLB} \Leftarrow$ number of TLB entries
$p_i \Leftarrow$ page to be obfuscated, read from RAM
$p_o \Leftarrow$ obfuscated page
$OTP_d \Leftarrow$ array of dynamic OTP
Function
for $k = 0$ to $N_{TLB} - 1$ do
  if $TLB[k].P$ is set then
    if $TLB[k].prot = NULL$ then
      Decrypt and validate $P_{conf}$
      if $D_{seq}, D_{cont}$ exist then
        $p_o = page\_unobfuscate(D_{seq}, D_{cont}, OTP_d, temp_i)$
        $p_i = temp_o$
      end if
      Generate new $D_{seq}, D_{cont}$
      Append it to $P_{conf}$
      $TLB[k].prot = \{S_{seq}, S_{cont}, D_{seq}, D_{cont}\}$
      Read the page in $p_i$
      $p_o = page\_obfuscate(D_{seq}, D_{cont}, OTP_d, p_i)$
      Write back $temp_o$
    end if
  end if
end for
```

The $TLB[k].Prot$ structure is the protected section of TLB entry and is cleared every time a new TLB entry is written. Hence, the function *dyn_obfuscate* is invoked on every TLB miss. If the page has already been subjected to dynamic obfuscation, *Arc3D* first performs the inverse operation (deobfuscation). It then generates new obfuscation configurations to perform dynamic obfuscation. This causes the dynamic obfuscation functions to be very short lived, i.e., changing on every page fault. It makes reverse engineering of π_D and C_D functions extremely unlikely. To ensure such a (π_D, C_D) refresh on every context switch, $TLB[k].Prot$ is cleared for all the entries whenever *start_prot_process* is called or a protected process is restored. A state register ST_i is allocated to the process and added to S_{auth}. The usage of this register is explained in Section 5.2.5. Availability of this register puts a limit on total number of protected processes active at any point in time in *Arc3D*. After the dynamic obfuscation is done, the process is started from PC_{start} as given by S_{auth}. The steps involved in *start_prot_process* are shown in Algorithm 5.

Algorithm 5. $start\_prot\_process$

```
1: Change to protected mode
2: Read $S_{auth}$ and validate
3: Read $S_{OTP}$ and validate
4: Generate $OTP_d$ and append to $S_{OTP}$
5: Clear $TLB[i].prot$ for all $i$
6: Call $dyn\_obfuscate$
7: Allocate $ST_i$ to the process and add it to $S_{auth}$
8: Set PC to $PC_{start}$
```

5.2.2 Memory Access

Once a process is started, it generates a sequence of instruction and data addresses. Like any high performance architecture, we assume separate TLBs, ITLB and DTLB, for instruction and data. Hence, the loading process explained earlier occurs parallelly in both ITLB and DTLB. The TLB is the key component of the obfuscation unit. The obfuscation functions are applied only during virtual to physical memory mapping. The address generation procedure is outlined in Algorithm 6. Two stages of F_{obf} are in the computation path for the physical address. This makes TLB latency higher than the single cycle latency of a typical TLB access. Hence, L1 caches of both instruction and data are made *virtually tagged* and *virtually addressed* to reduce the performance impact due to TLB latency. The L1 cache tags are extended with a *protection* bit, which is accessible only to *Arc3D*. This bit is set whenever the cache line is filled with data from a protected page. The access to protected cache-blocks is restricted only in protected mode. In order to have efficient context switching mechanism, we use a *write-through* L1 cache. Thus, at any point in time, L2 and L1 are in synch.

Algorithm 6. TLB_{xp} Access Function: $tlbxp\_access$

```
$v\_page \Leftarrow$ input virtual page address
$v\_block \Leftarrow$ input virtual block address
$p\_addr \Leftarrow$ output physical address
$k \leftarrow$ TLB index of hit and page exists in RAM
if $TLB[k].P$ is set then
  $p\_block = F_{obf}(D_{seq}, F_{obf}(S_{seq}, v\_block))$
else
  $p\_block = v\_block$ end if
$p\_addr = TLB[k].p\_page + p\_block$
```

TLB and L1 cache are accessed parallely. TLB is read in two stages. The first stage reads the normal portion of TLB and the second stage reads the extended and protected portion of TLB. This way the second stage access can be direct mapped and, hence, could be energy-efficient. If L1 access is a hit, then TLB access is stopped at $stage_1$. If L1 access is a miss, then TLB access proceeds as shown in the function *tlbxp_access*. In *Arc3D* L2 cache is *physically tagged and physically addressed*. Hence, no special protection is needed for the L2 cache. On an L2 cache access to an instruction in the middle of a cache-block, the relative intrablock sequence information is leaked to an observer adversary on the L1-L2 cache boundary. Given that for a 64 B cache-block size, there are 16 instructions whose sequencing information is open to exposure. One way to lessen this vulnerability is to have L1 cache only issue L2 cache-block addresses. The cache-block offset can be retained by the L1 cache for later decoding. Hence, the address traces visible at L1-L2 cache boundary will appear to be L2 cache-block address aligned. This would increase the latency but, as we discuss later, this increase will not be very high. Once the data is received from the L2 cache or memory, it is XORed with both OTP_d and OTP_s to get the actual content in plaintext which is then stored in an L1 cache line.

5.2.3 Execution

Arc3D has a set of protected registers (REG_p) to support protected process execution. This register set is accessible only in the protected mode. The protected process can use the normal registers to communicate with the OS and other unprotected applications. If two protected processes need to communicate in a *secure* way, then they have to use elaborate protocols to establish common obfuscation functions. Data sharing can also occur through a shared secret embedded into two applications by the software vendor in advance.

5.2.4 Interrupt Handling

Only instructions from a protected page can be executed in protected mode. Hence, any call to system services, such as dynamic linked libraries, requires a state change. Any interrupt causes *Arc3D* to go out of protected mode. Before transitioning to normal mode, *Arc3D* updates PC field in S_{auth} with the current PC. Thus, a protected process context could be suspended in the background while the interrupt handler is running in the unprotected mode. When the interrupt handler is done, it can execute *ret_prot_process* to return to the protected process. *Arc3D* takes the PC from S_{auth} and restarts from that point. This allows for efficient interrupt handling. But, from the interrupt handler, the OS could start other unprotected processes. This way, *Arc3D* does not have any overhead in a context switch from protected to unprotected processes. But, when the OS wants to load another protected process, the current protected process' context must be saved.

5.2.5 Saving and Restoring Protected Context

Arc3D exports *save_pro_process* API to save the current protected process context. This causes *Arc3D* to write $K_s\{REG_p\} + HMAC$ and S_{auth} into the memory given by the OS. The OS when restoring the *protected* process, should provide pointers to these data structures through *restore_prot_process*. *Arc3D* can be enabled to detect *replay* attacks by including an association of time with the savedcontexts. A set of OTP registers called state OTP registers are required within *Arc3D* for this purpose. These registers are the same size as K_s. The number of these registers depends on how many protected processes need to be supported simultaneously. The *start_prot_process* allocates a state OTP register ST_i. This association index ST_i is also stored within S_{auth}. Each instance of *save_prot_process* generates a state OTP value $OTP[ST_i]$ which is stored in ST_i. The saved context is encrypted with the key given by the XOR of K_s and $OTP[ST_i]$. Symmetrically, an instantiation of *restore_prot_process* first garners ST_i and K_s from S_{auth}. Then the key $OTP[ST_i] \oplus K_s$ is used to decrypt the restored context. This mechanism is very similar to the one used in all the earlier research, such as ABYSS and XOM.

5.2.6 Supporting Fork

In order to fork a protected process, the OS has to invoke *transfer_prot_process* API. This causes a new ST_i to be allocated to the forked child process. It then makes a copy of process context similar to *save_prot_process* handling. Thus, the parent and the child processes could be differentiated by *Arc3D*. The OS has to make a copy of S_{OTP} for the child process.

5.2.7 Exiting a Protected Process

When a protected process finishes execution, the OS has to invoke *exit_prot_process* API to relinquish the ST_i. This is the only resource that limits the number of protected processes allowed in an *Arc3D* system. Hence, *Arc3D* is susceptible to denial-of-service (DOS) kind of attacks.

5.2.8 Protected Cache

Arc3D has a protected direct mapped L2 cache of page size, i.e., $64KB$. This protected cache is used to obfuscate the second-order address sequences only for instructions, as temporal order doesn't have any meaning with respect to data. Whenever there is an IL1 miss in protected mode, *Arc3D* sends a request to $L2_{prot}$. Since $L2_{prot}$ is on-chip, the access latency will be small. We assume it to be 1 cycle. If there is a miss in $L2_{prot}$ then L2 is accessed. $L2_{prot}$ is also invalidated whenever a protected process is started or restored.

6 Discussion

6.1 Assumptions

In this section, we state and justify the underlying assumptions for *Arc3D*. The first and foremost of our assumptions is that every *Arc3D* processor has a unique identity (TPM's EK like identity). *Arc3D* device manufacturer can use various methodologies to embed the identity. Silicon Physical Random Functions (PUF) [20] have been proposed for this purpose. IBM's secure crypto-processor [21] provides a mechanism based on packaging for storing secrets within the processor environment. Xilinx [22] in its CPLD devices uses metal layers and dual access mechanisms to obfuscate the stored secrets.

The next issue is the extent of damage due to the exposure of the *Arc3D* identity secret. If an adversary is able to gain access to the stored secret, then all the programs that were distributed for that particular instance of *Arc3D* could be decrypted. Once the program plaintext is obtained, it can be executed in any *Arc3D* machine in unprotected mode. Hence, the ability to protect the stored secrets within the architecture is of paramount importance in *Arc3D* design. However, the programs distributed to and encrypted for other *Arc3D* platforms are not compromised by the exposure of the secrets of a given platform.

6.2 Attack Scenarios

In this section, we argue that *Arc3D* achieves our initial goals, namely, *copy-protection*, *tamper-resistance*, and *IP-protection*. Several attacks causing information leak in various dimensions could be combined to achieve the adversary's goal. These attacks could be classified into two categories—attacks that target *Arc3D* to manipulate its control or reveal its secrets. If the adversary is successful in either getting the stored secret (E_k^-) or in changing the control logic, the security assurances built upon *Arc3D* could be breached. But, these type of attacks have to be based on *hardware*, as there are no software control handles into *Arc3D*. There are several possible hardware attacks, like power profile analysis attacks and electromagnetic signal attacks. The scope of this paper is not to provide solutions to these attacks. Hence, we assume that *Arc3D* is designed with resistance to these hardware attacks.

The second type of attacks are white-box attacks. Such an attack tries to modify the interfaces of *Arc3D* to the external world, to modify the control. The guarantees that are provided by *Arc3D* to the software in protected mode of execution are 3D obfuscation for protected pages based on the unique identity per CPU. Protected mode of execution guarantees that the control is not transferred to any unauthorized code (which is undetected). *Arc3D* will fault when an instruction from an unprotected page or from a page that was protected with different K_s is fetched in protected mode. This will prevent attacks of the buffer overflow kind. 3D obfuscation provides us both IP-protection and tamper-resistance. IP-protection is achieved because at every stage of its life, the binary image is made to look different, hence reducing the correlation based information leaks to the maximum extent possible.

Correlation-based attacks are the ones where an adversary builds up information about the program behavior through repeated program executions. Such techniques [12] have been successfully used against commercial secure microcontroller DS5002FP [11]. In *Arc3D*, such attacks are

TABLE 1
Memory Hierarchy Simulation Parameters

Param	Alpha 21264 [24]	Intel XSCALE 80200 [25]
L1	64KB, 2 way, 64B, 3 cyc	32KB, 32-way, 32B, 3 cyc
ITLB/ DTLB	128 fully associative, 1 cyc	32 fully associative, 1 cyc
L2	1MB, 1 way, 16 cyc	256K, 8 way, 8 cyc
Memory	Lat 130 cyc, 4 bytes/cyc	Lat 32 cyc, 4 bytes/6 cyc
Peak B/w	7.1 GB/s	800 MB/s
Page sz	64KB	64KB

prevented, as the dynamic obfuscation functions are chosen at random for every process run, which prevents incremental information gain.

Tampering could be performed by many means. But, all of them have to modify the image of the process. Since every cache-block in every protected page potentially could have a different OTP, the probability that the adversary could insert a valid content is extremely small. Applications can obfuscate new pages that are created at run-time by designating them as protected. Applications can further maintain some form of Message Digest for sensitive data, because obfuscation only makes it harder to make any educated guess, while random modification of data is still possible. In the case of instructions, the probability that a random guess would form a *valid* instruction at a valid program point is extremely small.

Another form of tampering—splicing attack—uses valid cipher texts from different locations. This attack is not likely to succeed because every *cache-block* in every *page* has a unique OTP and every *page* has a unique address obfuscation function. This makes it hard for the adversary to find two *cache-blocks* with the same OTP. Another common attack is *replay attack*, where valid cipher text of a different instance of the same application is presented (replayed) to the CPU. As we discussed earlier, this attack is prevented by XORing K_s with a randomly generated OTP which is kept in the *Arc3D* state. This value is used as a key to encrypt the *protected* process' context. Thus, when restoring a protected context, *Arc3D* makes sure that both S_{auth} and saved context are from the same run.

When the adversary knows the internals of the underlying architecture, another form of attack is possible. This form of attack denies resources that are essential for the functioning of the underlying architecture. For example, XOM maintains a session table and has to store a *mutating register* value per session-id. This mutating register is used to prevent any replay attacks. This kind of architecture has an inherent limitation on the number of processes it can support, i.e., the scalability issue. Thus, an attacker could exhaust these resources and make the architecture nonfunctional. This kind of attack is possible in *Arc3D* as well on the state OTP register file. We could let the *context-save* and *context-restore* be embedded in the storage root of trust in a TPM like model. Such a model will allow *Arc3D* to perform in a stateless fashion which can prevent the resource exhaustion attacks.

TABLE 2
Simulation Results

XSCALE 80200					
Bench	IL1 Miss-rate	DL1 Miss-rate	ITLB Misses	DTLB Misses	%CPI Increase
bzip	0.0000	0.0225	2	256408	479
eon	0.0000	0.0020	10	12	0.145
gcc	0.0037	0.0510	28	110636	509
twolf	0.0000	0.0728	7	31	0.128
crafty	0.0009	0.0051	6	15627	73.4
gzip	0.0000	0.0231	3	1906	10.6
parser	0.0000	0.0354	5	50663	245

Alpha 21264					
Bench	IL1 Miss-rate	DL1 Miss-rate	ITLB Misses	DTLB Misses	%CPI Increase
bzip	0.0000	0.0185	2	113	0.12
eon	0.0000	0.0008	10	12	0.02
gcc	0.0019	0.0272	29	1804	0.97
twolf	0.0000	0.0508	7	31	0.01
crafty	0.0002	0.0123	6	33	0.02
gzip	0.0000	0.0125	3	1906	1.12
parser	0.0000	0.0210	5	1121	0.74
vpr	0.0000	0.0444	5	51	0.05

7 Performance Analysis

Since *Arc3D* seamlessly fits into the existing memory hierarchy as an extended TLB, the latency caused by *Arc3D* should be minimal. We used Simplescalar [23] Alpha simulator with memory hierarchy as shown in Fig. 2 to do the performance simulation. We did two sets of simulations with different latency parameters, Alpha 21264 and Intel XSCALE 80200 as shown in Table 1.

Three latencies are added by *Arc3D*, namely, extended TLB access, increased access time to L2 because of sending only block address to L2, and latency to read the pages and obfuscate them on every TLB miss. The first component gets absorbed in L1 cache access latency for both the systems, assuming that the extended TLB access increases the TLB access latency by two cycles. The major component is the reading time of *page* and writing it back to the memory. Since obfuscation is just an XOR operation, we can assume it takes one cycle. These facts, along with the assumption that these pages are transferred in and out of *Arc3D* at the peak memory bandwidth, lead to a latency increase of 12,000 cycles in the case of Alpha 2164 and 96,000 cycles in the case of XSCALE. The simulation was run with Spec2000 [26] benchmarks for two billion instructions by fast-forwarding the first 500 million instructions.

Table 2 shows that the performance impact on XSCALE 80200 memory hierarchy with higher number of TLB misses is greater than the impact on Alpha 21264 memory hierarchy. On Alpha 21264, the performance impact is less than 1 percent for most of the benchmarks.

8 Conclusion

Software obfuscation is a key technology in IP-protection. However, software only solutions (such as compiler transformations of control flow or insertion of redundant basic blocks or data structure transformations) often do not have robustness of crypto methods. Complete control flow obfuscation methods such as Cloakware [27] have the limitation that they cannot hide the correct control flow information from the prying eyes of the OS/end user. An additional weakness in these schemes is that observation of repeated dynamic execution often gives away the obfuscation secrets (such as control flow ordering or data structure sequencing).

We propose a minimal architecture, *Arc3D*, to support efficient obfuscation of both static binary file system image and dynamic execution traces. This obfuscation covers three aspects: address sequences, contents, and second-order address sequences (patterns in address sequences exercised by the first level of loops). We describe the obfuscation algorithm and schema, its hardware needs, and their performance impact. We also discuss the robustness provided by the proposed obfuscation schema.

A reliable method of distributing obfuscation keys is needed in our system. The same method can be used for safe and authenticated software distribution to provide copy-protection. A robust obfuscation also prevents tampering by rejecting a tampered instruction at an adversary desired program point with an extremely high probability. Hence, obfuscation and derivative tamper-resistance provide IP-protection. Consequently, *Arc3D* offers complete architecture support for copy-protection and IP-protection, the two key ingredients of software DRM.

Acknowledgments

This work is supported in part by an IBM PhD fellowship.

References

[1] "Business Software Alliance, Eighth Ann. BSA Global Software Piracy Study," *Proc. Trends in Software Piracy 1994-2002*, 2003.

[2] "Next-Generation Secure Computing Base," 2003, http://www.microsoft.com/ngscb.

[3] "Trusted Platform Module," *TCPA*, 2003.

[4] "TPM Design Principles—Version 1.2," Oct. 2003.

[5] S.R. White and L. Comerford, "ABYSS: An Architecture for Software Protection," *IEEE Trans. Software Eng.*, vol. 16, no. 6, pp. 619-629, June 1990.

[6] D. Lie, C.A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J.C. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 168-177, 2000.

[7] D. Aucsmith, "Tamper Resistant Software: An Implementation," *Information Hiding*, pp. 317-333, 1996.

[8] C.S. Collberg and C.D. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection," *IEEE Trans. Software Eng.*, vol. 28, no. 8, pp. 735-746, Aug. 2002.

[9] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 72-84, 2004.

[10] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431-473, 1996.

[11] "DS5002 Secure Microprocessor Chip," Dallas Semiconductor Mar. 2003.

[12] M.G. Kuhn, "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP," *IEEE Trans. Computers*, vol. 47, no. 10, pp. 1153-1157, Oct. 1998.

[13] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, Dept. of Computer Science, Univ. of Auckland, July 1997.

[14] C. Linn and S.K. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," *Proc. ACM Conf. Computer and Comm. Security*, pp. 290-299, 2003.

[15] E. Fredkin and T. Toffoli, "Conservative Logic," *Int'l J. Theoretical Physics*, vol. 21, nos. 3/4, Apr. 1982.

[16] R. Bennett and R. Landauer, "Fundamental Physical Limits of Computation," *Scientific Am.*, pp. 48-58, July 1985.

[17] T. Toffoli, "Reversible Computing," Technical Report MIT/LCS/TM151/1980, MIT Laboratory for Computer Science, 1980.

[18] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek, eds., pp. 31-39, 1994.

[19] HMAC: Internet RFC 2104, Feb. 1997.

[20] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon Physical Random Functions," *Proc. Ninth ACM Conf. Computer and Comm. Security*, pp. 148-160, 2002.

[21] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. vanDoorn, S.W. Smith, and S. Weingart, "Building the IBM 4758 Secure Coprocessor," *IEEE Computer*, vol. 34, no. 10, pp. 57-66, Oct. 2001.

[22] "Is Your FPGA Design Secure?" *XCELL J. Online*, 2003.

[23] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report 1342, Dept. of Computer Science, Univ. of Wisconsin-Madison, 1997.

[24] Z. Cvetanovic and R.E. Kessler, "Performance Analysis of the Alpha 21264-Based Compaq ES40 System," *Proc. Int'l Symp. Computer Architecture*, pp. 192-202, 2000.

[25] Intel 80200 Processor Based on Intel XSCALE Microarchitecture Datasheet, Jan. 2003.

[26] Specbench: SPEC 2000 Benchmarks, 2002.

[27] S. Chow, Y. Gu, H. Johnson, and V.A. Zakharov, "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs," *Proc. Int'l Security Conf. Lecture Notes in Computer Science*, vol. 2200, pp. 144-155, 2001.

Mahadevan Gomathisankaran received the BE degree in electronics and communcation engineering in 1998 from the Regional Engineering College (now known as the National Institute of Technology), Trichy, India and is currently working on the PhD degree in computer engineering at Iowa State University, Ames. His research interests are secure and DRM architectures, low power design, and logic design.

Akhilesh Tyagi received the BE degree (honors) in electrical and electronics engineering, in 1981 from the Birla Institute of Technology and Science, Pilani, India, followed by the MTech degree in computer engineering, in 1983 from the Indian Institute of Technology, New Delhi. He received the PhD degree in computer science from the University of Washington, Seattle, in 1988. He was an assistant professor in the Department of Computer Science at the University of North Carolina at Chapel Hill from August of 1987 to June of 1993. Subsequent to that, he was with the Department of Computer Science, and is now with the Electrical and Computer Engineering Department at Iowa State University, Ames. His research interests include VLSI complexity theory and low energy design, secure and DRM architectures and compilers. He is a member of the IEEE.