Architecture Support for 3D Obfuscation

Mahadevan Gomathisankaran and Akhilesh Tyagi
Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
{*gmdev,tyagi*}@*iastate.edu*

***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 DRM 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 3-D obfuscation through the use of well known cryptographic methods. The three dimensional obfuscation hides the address sequencing, the contents associated with an address, and the temporal reuse of address sequences such as in loops (or the second order address sequencing). The software is kept as an obfuscated file system image statically. Moreover, its execution traces are also dynamically obfuscated along all the three dimensions of address sequencing, contents and second order address sequencing. Such an obfuscation makes it infinitesimally likely that good tampering points can be detected. This in turn provides with a very good degree of tamper resistance. With the use of already known software distribution model of ABYSS and XOM, we can also ensure copy protection. This results in a complete DRM architecture to provide both copy protection and IP protection.**

***Index Terms*— Obfuscation, Digital rights management, Secure systems architecture.**

I. Introduction

DIGITAL rights management (DRM) deals with intellectual property (IP) protection and unauthorized copy protection. The IP protection is typically provided through a combined strategy of software obfuscation and tamper resistance. Thus these three properties become fundamental to DRM systems. DRM violations for software can result in either financial losses for the software developers or a loss of competitive advantage in a critical domain such as defense (for example when an aircraft lost in hostile territory contains embedded systems with critical IP). 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. Hence the primary distinction between the traditional security and DRM is that the focus shifts from the problem of *protecting the OS from an adversary* to the problem of *protecting an application program from the OS*.

Any software-only solution to achieve DRM seems to be inadequate. It leads to the classical meta-level inconsistencies encountered in classical software verification derived from Gödel's incompleteness theorem. 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 the Microsoft's next generation secure computing base (NGSCB) [6], 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) [7], to provide the SSC. The TPM is designed to provide such a root of trust

for storage, for measurement, and for reporting. It also supports establishment of a certified identity for the computing platform through its endorsement key pair. The certified identity (through public and private E_k pair) is the mechanism required for safe (trusted) distribution of the software from the vendor to the computing platform. This mechanism will be used to preserve the sanctity of the vendor provided static obfuscation. 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 [8]).

The paper is organized as follows. Section II describes the obfuscation problem and its interaction with the existing cryptographic solutions. Section III discusses earlier proposed research and their drawbacks. Section IV explains the basic building blocks of *Arc3D* and provides a high level overview. Section V provides operational details of *Arc3D* system. We describe various attack scenarios in Section VI Section VII gives the performance analysis of *Arc3D*. Section VIII concludes the paper.

II. The Problem

In this section, we describe the problem addressed in the paper. A software image is generated at the vendors site. This software image is then distributed to the customer through a transaction. Note that the software has to be generated in a standardized, well known, structure/format so that the OS can read it and load it. The control flow sequence of the generated software instructions ought to be understandable by the CPU, even if the program image is obfuscated. The customer, who could be an adversary, has access to everything outside the CPU chip boundaries, including memory of the system and the OS. Additionally, from copy protection perspective, the vendor would like to associate the software only to the machine/CPU covered by the purchase transaction. The software should run only in its original form, *i.e.*, as provided by the vendor. Its tampered forms should not be executable.

Based on these requirements, the specific attributes that need to be supported 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 reverse engineering. (*IP protection – static obfuscation*)
4) Any modification of the binary file should make the software unusable. (*IP protection – tamper resistance*)
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 sharing of a secret on a secure encrypted channel. This is a well analyzed problem and solutions like Pretty Good Privacy (PGP) exist. The two unknown parties establish trust through a common trusted third party, namely, a Certification Authority (CA). The two parties then share their public keys (RSA) and hence can send messages readable only by the intended audience. This approach has been used in almost all the earlier research dealing with copy protection ([4], [2]). We will be also deploy a similar approach. Note that the TPM specifications require the establishment of an endorsement key pair which is also registered with a CA. Furthermore, any number of attestation key pairs can be created to support trust between the CPU (with this TPM) and other clients (such as software vendors) which can also be registered with a certifying authority. The TPM specified protocol for creating an attestation identity also creates a shared secret between the TPM (or the corresponding CPU) and the client (software vendor). We expect to be able to use this capability of TPM or a similar protocol for copy protection.

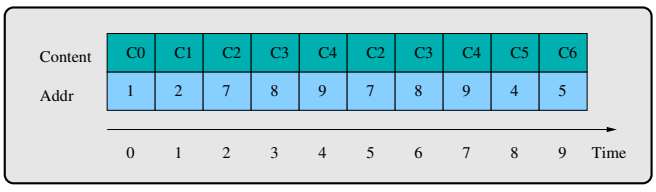
The third problem requires prevention (minimization) of information leak from the static binary file/image. This could be viewed as the problem of protecting a message in an untrustworthy channel. One possible solution is to encrypt the binary file (the solution adopted by XOM [2] and ABYSS [4]). General asymmetric encryption is a more powerful hammer than necessary for this problem which is also less efficient computationally. An alternative approach would recognize that the binary file is a sequence of instructions and data with an underlying structure. Static obfuscation [9], [10] attempts to achieve this effect. In fact, when the whitebox behavior of the obfuscated program $\mathcal{T}(\mathcal{P})$ is not differentiable from its blackbox behavior, for all practical purposes $\mathcal{T}(\mathcal{P})$ provides all the protection of an encrypted version of

$\mathcal{P}$. Our solution to this is to use 3D (three dimensional) obfuscation that obfuscates the address sequencing, contents, and second order address sequencing.

Fourth problem requires the binary file to be tamper resistant. This could be interpreted as any modifications to the binary file should be detectable by the hardware. Message Digest, which is a one-way hash of the message itself, solves this problem. This once again is a generic solution which is applicable to any message transaction (which does not use the special properties of binary file). 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 (such as 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 these three dimensions have to be protected in order to prevent any information leakage. This holds true even for data. The three dimensions of information space could be named as address, content and time order. Figure 1 shows such an arbitrary sequence of accesses.

Fig. 1
ARBITRARY ACCESS SEQUENCE



III. EARLIER RESEARCH

A. ABYSS

ABYSS [4] is an architecture for protecting the application software. It can be used as a uniform security service across a range of computing systems. The system contains both *protected* and *unprotected* processes. Protected processes are executed in a *protected processor*. Protected processor constitutes a minimal, but complete, computing system. It has sufficient memory to store protected parts of the application. It also has non-volatile storage space to store *Rights-To-Execute* (RTE) attributes of an application. There exists a *supervisor process* which ensures the logical and procedural security of the protected processor.

Supervisor process handles the decryption of protected process from disk and its loading into RAM. It also isolates the applications from each other, and from their unprotected parts. "Rights to execute" contains the privileges of the application and a "Key" to decrypt the application. *One-Use* tokens are used to authorize the installation of RTE. The encryption model used is shown in Box 1. Where, K_A is the Application key, decided by the software vendor and can be unique for each copy or common for multiple copies. K_S is the Supervisor key and is the shared secret between software vendor and hardware manufacturer. Drawbacks of ABYSS include non-scalability and unexplained OS interactions.

$$\begin{array}{c} K_A\{\ Protected\ part\ of\ application\ +\ Tok\ \} \\ K_S\{\ RTE\ +\ K_A\ \} \end{array} \quad (1)$$

B. TrustNo1 Cryptoprocessor

TrustNo1 [5] proposes hardware, firmware, OS and key management mechanisms necessary to apply the cryptoprocessor concept in multitasking OS systems. It contains a protected on-chip memory key table that can store segment keys. This key table can only be accessed by firmware that is also stored in protected on-chip memory. Segment descriptors are extended by a new field, which contains the index of the key used to decrypt it. Each cache line is encrypted with the segment's key and some of the MSB bits of cache line address. This makes each key unique and hence prevents the cipher instruction search attack. The memory manager grants any access only if it originates from an instruction in

the same segment. This prevents even OS from reading or modifying the cleartext of an encrypted segment and from calling parts of the code in an uncontrolled fashion. Box 2 lists the special operations supported by the cryptoprocessor.

$$\begin{array}{lcl} save\_state & \rightarrow & store\ CPU\ state \\ restore\_state & \rightarrow & restore\ CPU\ state \\ transfer\_state & \rightarrow & copy\ current\ CPU\ state \\ supervisor\_call & \rightarrow & execute\ system\ calls \end{array} \quad (2)$$

C. XOM

XOM [2] is based on 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. All levels of memory (L2, L1, Registers) are tagged with the session key ID, specific to the process that is running. XOM stores session keys in the key table. The basic operations exported by XOM are *enter_xom, exit_xom, mv_to_null, mv_from_null, save_secure, restore_secure, load_secure, store_secure, load_from_null,* and *store_to_null*. Using these an OS can manage any protected process and all the normal operations in an XOM node, except *fork* of a protected process.

D. HIDE

HIDE [3] is an extension of XOM. It points out the fact that XOM does not protect the third dimension of the information space, *i.e.*, the time order of the address trace. Hence even if the instructions (and data) themselves are obfuscated (through encryption in XOM), the address trace gives the adversary power to deduce the control flow graph (CFG). HIDE further argues that the information obtained from the CFG could lead to serious security breaches of the software. HIDE provides a solution to this address bus leakage problem consisting of chunk-level protection with hardware support and a flexible interface. Compiler optimizations and user specifications could be further utilized to deploy the underlying hardware solution more efficiently to provide better security guarantees.

E. Analysis

Almost all of the earlier research except HIDE does not hide the temporal sequencing of memory accesses. Another shortcoming of these solutions is that they do not exploit the software specific properties. All of them use encryption and message digest as a means to make the software tamper resistant. However if both the temporal instruction sequences and instructions themselves are obfuscated, the adversary will have to reverse-engineer both of these attributes in order to tamper with the software. If both these attributes (instruction sequence and instruction content) are strongly protected then the adversary needs to perform more work than the additive work of breaking the protection for each individual attribute.

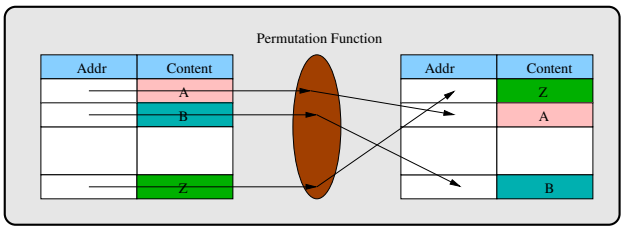
The solution proposed by HIDE to prevent information leak through the address and memory bus is weak. This is because the adversary can see the contents of the memory both before and after an address permutation. It is possible because the encryption function applied to the contents is not address dependent. 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. Figure 2 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 1024 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.

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.

IV. Proposed Architecture: Arc3D

The overall proposed architecture of Arc3D is shown in Figure 3. The main affected components of the

Fig. 2
WEAKNESS OF HIDE APPROACH



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.

A. Obfuscation Schema

A program is obfuscated both statically and dynamically. The objective of obfuscation is to permute the address sequence and to hide the contents (so that the mapping from an address A_i to its contents C_{A_i} is not obvious). Let V denote the virtual address sequence $\{0, 1, 2, \ldots, 2^{32}-1\}$ for a 32-bit architecture. The classical static binary image layout maps instructions in the order: $I_0, I_1, \ldots, I_{2^{32}-1}$. In other words, contents I_j are mapped in the virtual address sequence $j \in V$. A static address permutation function π_S can be applied to the binary image so that the instructions in the static binary image appear in the following order: $I_{\pi_S^{-1}(0)}, I_{\pi_S^{-1}(1)}, I_{\pi_S^{-1}(2)}, \ldots, I_{\pi_S^{-1}(2^{32}-1)}$. The static address permutation function disperses the jth instruction in the virtual address sequence V, I_j, into the static sequence number $\pi_S(j)$.

The contents also need to be obfuscated. A content obfuscation function $C_S(I_j, j)$ transforms the original contents I_j. Note the dependence of the content obfuscation function on the address/sequence number of the instruction j as well. This makes sure that the contents I (an instruction encoded as I) will look different when mapped to two distinct sequence numbers j and k. Hence the program will appear to be in the following sequence after static obfuscation has been applied by the software vendor: $C_S(I_{\pi_S^{-1}(0)}, 0), C_S(I_{\pi_S^{-1}(1)}, 1), C_S(I_{\pi_S^{-1}(2)}, 2), \ldots, C_S(I_{\pi_S^{-1}(2^{32}-1)}, 2^{32}-1)$.

The dynamic execution of the program will need to know both the content obfuscation function C_S and the address permutation function π_S. When an instruction in the virtual address sequence $j \in V$ needs to be fetched, the processor would have to issue an address $\pi_S(j)$. Let memory return $M[\pi_S(j)]$ in response to this read. Now the processor would have to deobfuscate the contents as $C_S^{-1}(M[\pi_S(j)], j)$. This schema sums up (and generalizes) most permutation based obfuscation schemes. The problem with this approach though is that an adversary watching the address bus can infer the correct address sequence for the instructions since both j and $\pi_S(j)$ are known for many (or all) instantiated addresses $j \in V$. Goldreich et al. [11] provide a theoretical solution to this problem which is applicable to a software-hardware (SH) package. Specifically, it seems to assume a lightweight, embedded operating system which is not necessarily controlled by the owner of the SH-package.

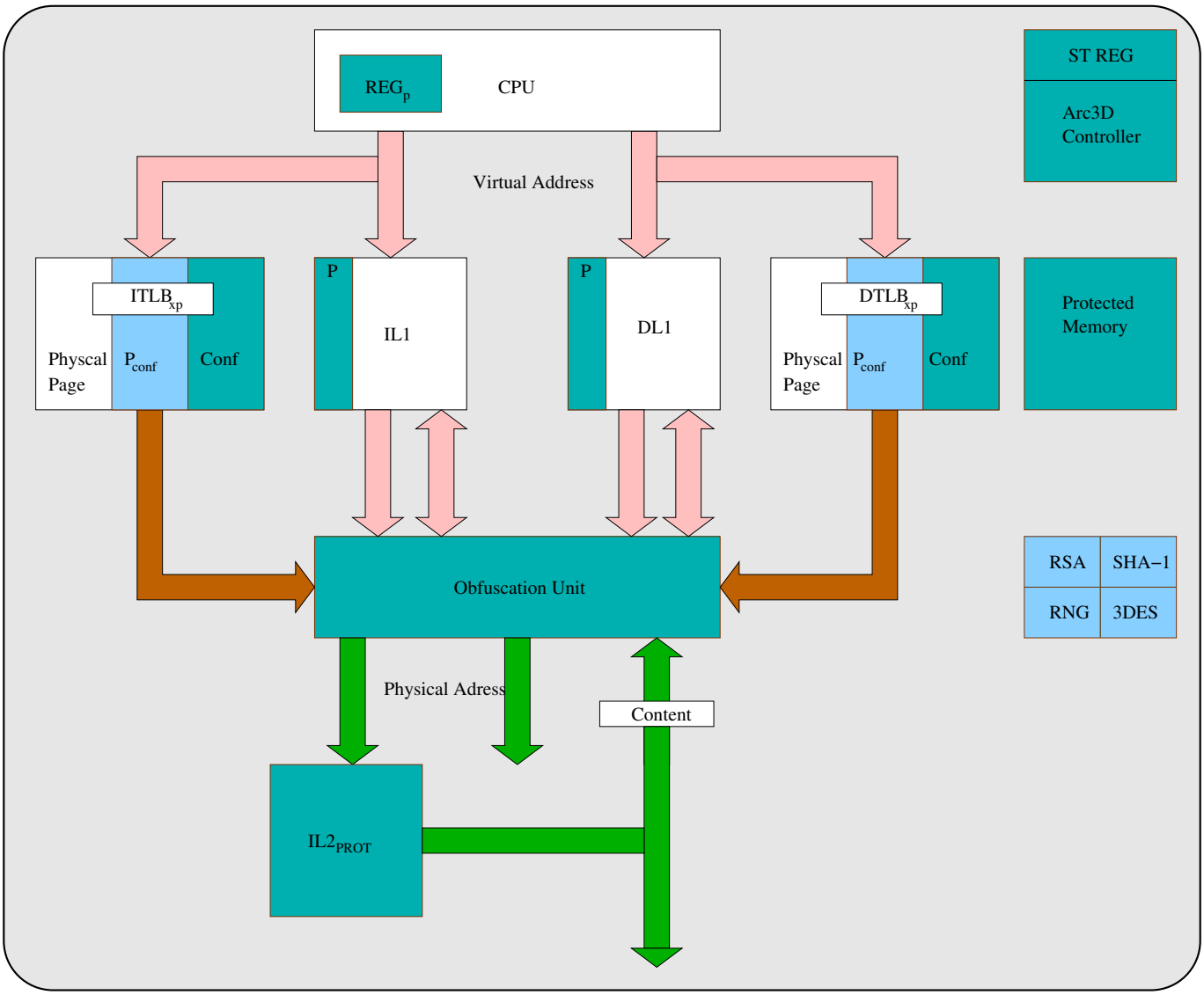
We propose to perform yet another level of dynamic obfuscation so that address bus visible address sequence is yet another permutation of the virtual address sequence. Another pair of address permutation and content obfuscation functions which are dynamically chosen, π_D and $C_D(I_j, j)$, help achieve dynamic obfuscation. Hence the static image sequence $C_S(I_{\pi_S^{-1}(0)}, 0)$, $C_S(I_{\pi_S^{-1}(1)}, 1)$, $C_S(I_{\pi_S^{-1}(2)}, 2)$, $\ldots$, $C_S(I_{\pi_S^{-1}(2^{32}-1)}, 2^{32}-1)$ is actually loaded in the memory as $C_D(I_{\pi_D^{-1}(0)}, 0)$, $C_D(I_{\pi_D^{-1}(1)}, 1)$, $C_D(I_{\pi_D^{-1}(2)}, 2)$, $\ldots$, $C_D(I_{\pi_D^{-1}(2^{32}-1)}, 2^{32}-1)$. Hence it is important that the loading and storing (specifically virtual address translation) function be taken away from the operating system and be part of a trusted component within the architecture. This trusted component is also responsible for guarding the program secrets C_S, C_D, π_S, π_D.

B. Overall Schema

As stated earlier, Figure 3 shows the global floor-plan for the proposed architecture. The shaded areas are the additional components of *Arc3D* over the base architecture. Shading hues also indicate the access rights as follows. Whereas 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, and it is achieved by modifying the virtual address translation path - translation look aside buffer (TLB) - of the base architecture. The TLB in addition to holding the virtual address to physical address mapping, page table entry (PTE), has the obfuscation configuration (P_{conf}). This

Fig. 3
OVERALL SCHEMA OF *Arc3D* ARCHITECTURE



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 the same in decrypted form in $Conf$ section of TLB_{xp}. This section of TLB is updated whenever a new PTE is loaded into the TLB_{xp}. *Arc3D* assumes parallel address translation paths for data and instructions, and hence Figure 3 shows DTLB and ITLB separately.

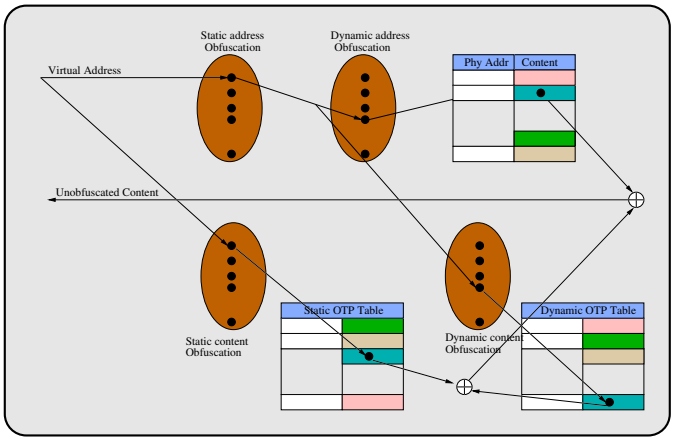
The address translation for the protected process occurs in the obfuscation unit, which is expanded in Figure 4. Sections IV-D and IV-E explain in detail the address sequence and content obfuscation algorithms respectively. *Arc3D* uses same logic for both static and dynamic obfuscations, and the basis of these obfuscations is the permutation function which is explained in Section IV-C. *Arc3D* has a protected L2 cache, which is accessible only to a protected process, thus providing temporal order obfuscation.

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 the protected process
2) *exit_prot_process*: Free the protected resources allocated for this process
3) *ret_prot_process*: Return to protected process from an interrupt handler
4) *restore_prot_process*: Restore a protected process after a context switch
5) *transfer_prot_process*: Fork a protected process

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

Fig. 4
ADDRESS AND CONTENT OBFUSCATION



C. 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. Figure 5 shows one such schema for permutation of 10 address bits (specifying a page consisting of 1024 cache blocks). Before explaining the blocks of Figure 5, we observe that there are $\left(2^{2^n}\right)^n$ possible functions implemented in a $n \times n$ look up table (LUT) or n n-LUTs. But only a subset of them are bijective. We wish to implement only reversible (conservative) gates ([12], [13]) with LUTs.

A conservative gate does not lose any information in going from its inputs to outputs. We should be able to infer the input values uniquely by observing the output bits of such a gate. Thus a reversible gate needs to have as many outputs as inputs. Both Fredkin [12] and Toffoli [14] have defined classes of reversible gates.

Definition 1. *Toffoli gate, Toffoli(n,n)(C,T), is defined over a support set* $\{x_1, x_2, \ldots, x_n\}$ *as follows. Let the control set* $C = \{x_{i1}, x_{i2}, \ldots, x_{ik}\}$ *and the target set* $T = \{x_j\}$ *be such that* $C \cap T = \emptyset$*. The mapping is given by*

$$Toffoli(n,n)(C,T)[x_1, x_2, \ldots, x_n] =$$

$$[x_1, x_2, \ldots, x_{j-1}, z, x_{j+1}, \ldots, x_n]$$

where $z = x_j \oplus (x_{i1} \wedge x_{i2} \wedge \ \ldots \ \wedge x_{ik})$.

Definition 2. *Fredkin gate, Fredkin(n,n)(C,T), is defined over a support set* $\{x_1, x_2, \ldots, x_n\}$ *as follows. Let the control set* $C = \{x_{i1}, x_{i2}, \ldots, x_{ik}\}$ *and the target set* $T = \{x_j, x_l\}$ *be such that* $C \cap T = \emptyset$*. The mapping is given by*

$$Fredkin(n,n)(C,T)[x_1, x_2, \ldots, x_n] =$$

$$[x_1, x_2, \ldots, x_{j-1}, p, x_{j+1}, \ldots, q, \ldots, x_n]$$

where $k = x_{i1} \cdot x_{i2} \cdot \cdots \cdot x_{ik}$, $p = (x_j \cdot \bar{k}) + (x_l \cdot k)$, *and* $q = (x_j \cdot k) + (x_l \cdot \bar{k})$.

We use *Toffoli(5,5)* gates with 5-input bits and 5-output bits in our scheme as shown in Figure 5. However, we could easily replace them by *Fredkin(5,5)* gates. The domain of configurations mappable to each of these LUTs consists of selections of sets T and C such that $T \cap C = \emptyset$. For a support set of 5 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 estimate this redundancy later in this section.

The exchanger blocks shown in Figure 5 perform *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. Some other more 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 [15]). In addition to 4 *input bits*, each LUT will also have

Fig. 5
RECONFIGURABLE BIJECTIVE OBFUSCATION UNIT

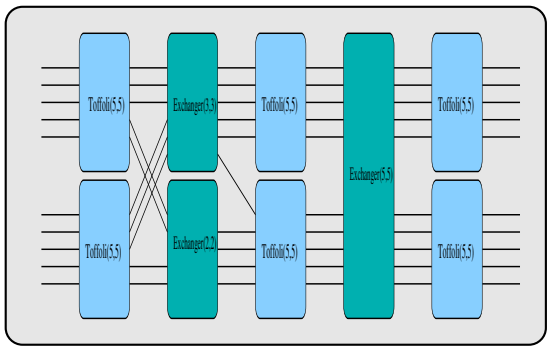
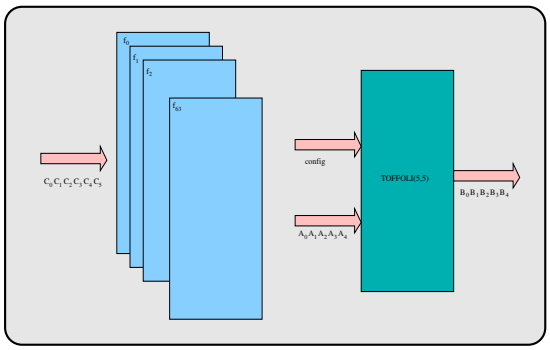


Fig. 6
CONFIGURATION SELECTION FOR EACH LUT



6 *configuration bits* to choose one of the 55 configurations (assuming some configurations are repeated to fill the 64 locations), as shown in Figure 6. Each of the *exchanger* blocks also requires 1 configuration bit. Thus a total of 39 configuration bits are needed by the reversible logic of Figure 5.

Estimating Redundancy in Configurations: The most reasonable and efficient way to generate configurations is to generate each configuration bit independently and randomly. However this process may generate two configurations that represent the same mapping (from incoming block# to the outgoing block#). Such aliasing reduces the diversity of the address mapping functions making them more predictable to the adversary. We capture the degree of this aliasing with the concept of *redundancy level* of a reconfigurable obfuscation circuit. The redundancy level can be defined as the fraction of 2^{39} configurations that alias (generate a repeated, non-unique mapping function).

We assessed the redundancy level of the address permutation schema in Figure 5 through the following setup. We simulated this FPGA circuit with 2^{20} randomly generated configurations. For each of these configurations, we derived the corresponding bijective function by exercising all the 10-bit inputs sequences. Each unique bijective function was stored. When a bijective function f_i from a new random sequence from the 2^{20} runs is encountered, it is compared against all the stored bijective functions that have already been generated. At the end of 2^{20} runs, we end up with $k \leq 2^{20}$ unique functions f_i for $0 \leq i \leq k$ and their redundancy count r_i (function f_i occurs in r_i of the 2^{20} runs). The *redundancy level* is computed as $[\sum_{r_i>1} 1]/2^{20}$. We repeated this experiment several times in order to get a statistical validation of our experiment. All the values are listed in Table I.

TABLE I
REDUNDANCY ESTIMATION: # OF RNDM CONFIGS = 2^{20}

Random Seed	# of Redundant Functions	Avg % Redundancy	99% CI
89ABCDEF	3359	0.320	0.3058 to 0.3342
11223344	3409	0.325	0.3107 to 0.3393
12345678	3441	0.328	0.3136 to 0.3424
34567890	3417	0.325	0.3107 to 0.3393
789012345	3469	0.330	0.3156 to 0.3444
8901234567	3460	0.330	0.3156 to 0.3444
56789012345	3460	0.330	0.3156 to 0.3444

This experiment can be modeled as a random experiment where we have $N(=2^{39})$ balls in a basket which are either *red*(=*redundant*) or *green*(=*non-redundant*). We need to estimate the number of red balls in the basket by picking $n(=2^{20})$ balls where all the balls are equally likely. We define a random variable X such that $X = 1$ if the chosen ball is *red* and $X = 0$ otherwise. The mean of such a random variable is nothing but the redundancy level. We see from Table I that the mean is close to zero $(\approx 0.3\%)$ and hence the variance is equal to the mean. Using the variance and mean we estimated the 99% confidence interval of the mean of X, *i.e.*, the average redundancy level of reconfigurable obfuscation circuit. From the table, it is clear that with probability 0.99 the average percentage of redundant configurations will lie within 0.3058 to 0.3444, *i.e.*, only 3 out of 1000 randomly generated configurations will be redundant.

D. Obfuscating the Sequence

We can use the reconfigurable unit defined in Section IV-C to achieve sequence obfuscation. The point to be noted is, even though we have shown the circuit for 10 *bits*, the methodology is applicable to an arbitrary number of address bits. But to have any reasonably complex permutation space we need at least 10 address

bits. Another reason to use 10 *bits* is due to the structure of the software. Software (both instruction and data) as we know is viewed by the architecture in various units of sizes (granularities). While residing in RAM it is viewed in units of *pages* and while residing in cache it is viewed in units of *blocks*. This allows us to obfuscate the sequences of these units. Hence we obfuscate the sequence of *cache blocks* within a page. The limitation of obfuscating within a *page* level is brought in by the fact that page management is done by the OS and any obfuscation that crosses *page* boundary has to expose the permutation function (π) to the OS, which itself causes information leak. This is also the reason why we can't obfuscate the sequences of pages. In the other direction, permuting the sequences of sub-units 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 64KB with 64B *cache blocks*, as is the case with Alpha-21264, we get 1024 *cache blocks* per page, *i.e.*, 10 *bits* of obfuscation.

E. 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 structured nature of software comes in handy for us once again. We can consider a software as a sequence of fixed sized messages, *i.e.*, *cache blocks*. Thus if we have unique OTPs for each one of the cache blocks in the software, it is completely protected. However maintaining that many OTPs is highly inefficient. Moreover we at least have to guarantee that every *cache block* within a *page* has a unique OTP. This is to overcome the weakness related to HIDE as explained in Section III, Figure 2. 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 achieve the unique cache block OTP per page property, one option is to generate a random OTP mask for each cache block for each page. A more efficient solution, however, is to pre-generate 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 the unique cache block OTP per page property. This simplifies the hardware implementation of content obfuscation unit as well since each page is processed uniformly in this unit except for for the π_c function. Hence a software image will need to provide a page of OTPs which will be used for all the pages for this software. Additionally, it also needs to specify a unique mapping function π_c per page. Since we already have the reconfigurable permutation logic of Section IV-C 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. As with any OTP related encryption the problem is not in generating the OTP but that of sharing the OTP. While estimating the complexity of the system we need to take this into consideration. Although we use XOR as our OTP function, it can be easily replaced with any other *bijective* function. The static obfuscation part of Figure 4 explains these operations clearly.

F. 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 ith instruction in the loop body in the jth 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 sequence order relative to 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 learnt 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 CPU is naturally obfuscated due to various factors like *caching*, *prefetching*, and various

other *prediction* mechanisms aimed at improving the performance. But these architecture features are also controllable, directly or indirectly, by 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* non-existent. To overcome such OS directed attacks, it is sufficient to have a reasonably sized *protected cache* in the architecture which is *privileged* (accessible to secure processes alone). We expect that *page* sized, in our case 64KB, *cache* should be sufficient to mask the effects of loops. Encrypted or content-obfuscated *cache blocks* already obfuscates CFGs (within the cache block) as 64B can contain 16 instructions if we assume instructions of length 32-bits.

V. Arc3D in Operation

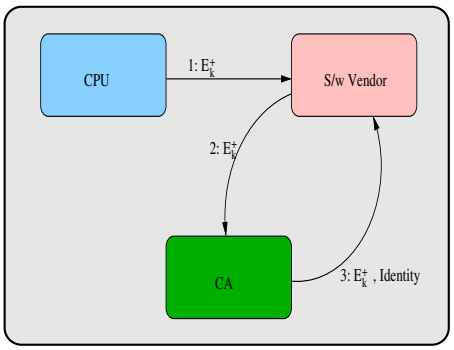
We have developed and described all the building blocks of *Arc3D* in Section IV. In this section, we explain its operation with respect to the software interactions in detail, from software distribution to management of protected process by OS using APIs provided by *Arc3D*.

A. Software Distribution

Arc3D provides both *tamper resistance* and *IP protection* with obfuscation. Hence, a software vendor should be able to obfuscate the static image of the binary executable. Moreover, a mechanism to distribute the de-obfuscation function 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 information can be transferred to *Arc3D*.

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* $(E_k{}^+, E_k{}^-)$. 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 CPU vendor's identity and other information (such as model number, part number, etc.).

Fig. 7
THREE PARTY TRUST MODEL



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 well known 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 Figure 7.

An important point to note here is that trust establishment and 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 [7] model.

2) Binary Image Generation: 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, minimum supported *page* size. Software vendor generates the binary file targeted at the appropriate cache block and page sizes. It generates two sets of random configuration bits 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, $OTP[b_i]$ 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. We assume that the software vendor has access to an *Arc3D*-enabled CPU with an obfuscation unit or its algorithm. Note that the obfuscation algorithm itself could be made public as is the case with any encryption algorithm. Only the configuration bits that specify various obfuscation mapping functions need to be guarded.

This kind of obfuscation can be applied to any binary

Fig. 8

PAGE OBFUSCATION FUNCTION

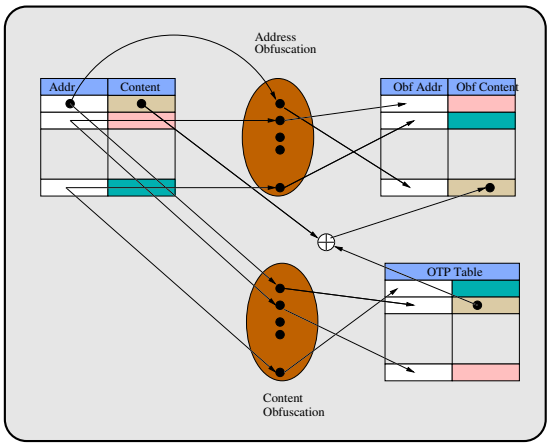


image as this does not cross *page* boundary, which is the unit of operation for the OS. It does not cross *cache block* boundary either which is the unit of operation for the architecture. But the software vendor could restrict the distribution to only those machines which have a certain minimum *cache block* size and *page* size, as both these parameters affect the strength of obfuscation. A suggested minimum for these parameters is $64B$ and $64KB$ respectively. Since this obfuscation is performed on static binary images, which do not have any run-time information, we call this *static obfuscation*. The basis of static obfuscation is a page obfuscation function (*page_obfuscate*) which takes an input page, an OTP page, configuration selection bits for both address sequence and content, and produces an output page. The outline of this algorithm is shown in Algorithm-1 and in Figure 8. The algorithm for *static obfuscation* is shown in Algorithm-2.

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

Software vendor generates S_{seq}, configuration selection for sequence obfuscation (corresponding to π_S), for every page to be protected and S_{cont}, configuration selection for content obfuscation (corresponding to π_{c_s}), and uses *page_obfuscate* to obfuscate the page, and associate the configuration information with the page. This is explained in Algorithm-2. Even for pages which are not loaded, 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 sections containing the configuration information. These configuration bits 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. Then each page level configurations S_{seq} and S_{cont}, are encrypted with this K_s. An HMAC [16] of this encrypted configuration is also generated. HMAC is a keyed hash which will allow Arc3D to detect any tampering of the encrypted configurations. Let P_{conf} represent encrypted configuration bits and its HMAC and let S_{conf} represent this section containing P_{conf} of all the pages. The new binary format should carry encrypted configuration bits and its HMAC for every protected page. The page containing the page block OTPs also needs to be stored. This page (containing OTP) is also encrypted with K_s. Its HMAC is computed 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 2

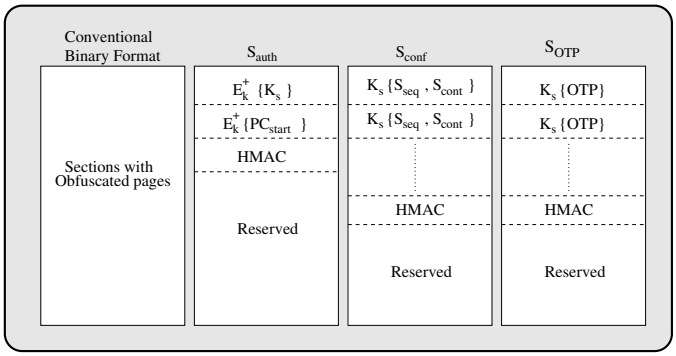
STATIC OBFUSCATION FUNCTION: *stat_obfuscate*

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

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 sending it with the software to the CPU. Now only the CPU with the private key ${E_k}^-$ can decrypt the distributed image to extract K_s. The entry point of the software 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 Figure 9.

Fig. 9
EXTENDED BINARY FORMAT



As we have argued earlier, the obfuscation process makes the software tamper resistant. In order to tamper the software in an undetectable and advantageous manner, the adversary must know the OTP. The probability of guessing an OTP is very small as we use $64B$ length OTPs. Note that the encrypted contents of blocks from a different page do not leak any information about the OTPs. This is so because the ith OTP, $OTP[b_i]$ is applied to different blocks $\pi_S^P(i)$ and $\pi_S^{P'}(i)$ in two different pages P and P'. Hence this form of obfuscation is at least as strong as the function guarding the configuration bits. The symmetric encryption with K_s guards the configuration bits. The symmetric key K_s can be of any arbitrary length. Its length will have little impact on the *Arc3D* performance(as shown later). The complete algorithm for software distribution step is shown in Algorithm-3.

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

B. Management of Protected Process

In this section we will explain, in detail, how OS uses the API provided by *Arc3D* controller to manage a protected process. We will also see how seamlessly it can be integrated with the existing systems while providing the guarantees of *tamper resistance* and *copy protection*.

1) Starting a Protected Process: *Arc3D* has two execution modes, (1) protected and (2) normal, which are enforced without necessarily requiring the OS cooperation. This is similar to the *nexus* mode of NGSCB [6]. When the OS creates a process corresponding to a protected software, 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 configuration bits. The decision whether to extend page table entry (PTE) with these configuration bits is OS and architecture dependent. We consider an architecture in which TLB misses are handled by the software and hence OS could 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 (a page which has been obfuscated). Additionally, for every protected page, OS has to set the protected mode bit P. This will be used by the architecture to decide whether to use obfuscation function or not. 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 $start\_prot\_process$ primitive to start the protected process execution. This causes *Arc3D* to transition to *protected* mode. *Arc3D* decrypts S_{auth} and checks its validity by generating the 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 [3], the address sequence information suffices to reverse engineer the IP without even knowing the actual instructions. *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, one to obfuscate the sequence of *cache block* addresses within a *page*, and the other obfuscating the contents of *cache block* within a page. The obfuscation engine for implementing dynamic obfuscation is very similar to the static obfuscation engine. Similar data structures could be used to save the configuration bits. Once *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 OTP_s and OTP_d at the same point in time. It reads S_{OTP} and decrypts OTP_s, and 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 S_{OTP} has reserved space for OTP_d and its HMAC in advance.

Arc3D then scans the TLB and for every protected page that has been loaded in main memory (RAM). It validates P_{conf}. It then generates D_{seq} and D_{cont} configuration bits (corresponding to π_D and π_{c_d}) for each one of those pages and appends them to P_{conf}. TLB_{xp} which has been extended to have 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 $S_{seq}, S_{cont}, D_{seq}, D_{cont}$ configuration bits, 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.

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, it 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 V-B.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 high level steps involved in *start_prot_process* are shown in Algorithm-5.

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

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}

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 for instruction and data. Hence the loading process explained earlier occurs parallely 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 logic is explained 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 cache blocks with protected bit set 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 explained 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. One point to note here is that the physical addresses generated by TLB_{xp} are cache block boundary aligned (integer multiple of cache block size, 64B in our example and hence with 6 least significant 0s). This is because, as we know, there could be many instructions within a cache block, and the instruction sequence within the cache block is not obfuscated. Hence, an exposure of the actual addresses falling within a cache block leaks information. This leads to the cache block aligned address restriction. 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 *XOR*ed with both OTP_d and OTP_s to get the actual content in plaintext which is then stored in an L1 cache line.

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 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. Since K_s is known to the application itself, it can modify the P_{conf} such that it shares the same configuration functions with the other processes (if need be).

4) Interrupt Handling: Only instructions from a protected page can be executed in protected mode. Hence any services, such as dynamic linked libraries, require a state change. Any interrupt causes the *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 OS wants to load another protected process the current protected process' context must be saved.

5) Saving and Restoring Protected Context: Arc3D exports *save_prot_process* primitive 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*. But this model has a flaw, as we don't have any association of time with respect to the saved context. Hence *Arc3D* cannot detect if there is a *replay* attack. *Arc3D* needs to keep some state which associates the program contexts with a notion of time. A set of OTP registers called state OTP registers are required within *Arc3D* for this purpose. These registers will have 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, for this protected process. 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 state OTP register. The saved context is encrypted with the key given by the XOR of K_s and $OTP[ST_i]$. On the other hand, an instantiation of *restore_prot_process* first garners ST_i and K_s from S_{auth}. 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.

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

7) Exiting a Protected Process: When a protected process finishes execution, OS has to invoke *exit_prot_process* API to relinquish the ST_i (state OTP register) allocated to the process currently in context. This is the only resource that limits the number of protected process allowed in the *Arc3D* system. Hence *Arc3D* is susceptible to denial-of-service (DOS) kind of attacks.

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 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.

VI. 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, Electro magnetic 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 software in protected mode of execution are 3D obfuscation for protected pages, and 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 buffer overflow kind of attacks. 3D obfuscation provides us both IP protection and tamper resistance. IP protection is achieved because at every stage of its life, the binary is made to look different, hence reducing the correlation based information leaks to the maximum extent possible.

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 possible 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 passed (replayed). 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's 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 is denial of resources, which 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 non-functional. This kind of attack is

possible in *Arc3D* as well on the state OTP register file. We could let the context save and 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.

VII. 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 [20] Alpha simulator with memory hierarchy as shown in Figure 3 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 II.

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 2 cycles. The major component is the reading time of *page* and writing it back in the memory. Since obfuscation is just an XOR operation, we can assume it occurs in single cycle latency for every XOR operation. These facts along with the assumption that these pages are transferred in and out of *Arc3D* at the peak memory bandwidth, we get 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 [19] benchmarks for 2 Billion instructions with fastforwarding the first 500 million instructions.

From Table III we see that this solution has greater impact on performance in the case of XSCALE 80200 memory hierarchy where TLB misses are more, but with Alpha 21264 the performance impact is less than 1% for most of the benchmarks.

VIII. 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 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 repeated dynamic execution observation 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.

References

[1] Business Software Alliance, 8th annual BSA global software piracy study. Trends in software piracy 1994-2002, 2003.

[2] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. *Architectural support for copy and tamper resistant software*, In Proceedings of ASPLOS 2000, pages 168-177.

[3] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. *HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus*, In Proceedings of ASPLOS 2004.

[4] Steve R. White and Liam Comerford. *ABYSS: An Architecture for Software Protection*, IEEE Transactions on Software Engineering, Vol. 16, No. 6, June 1990, pages 619-629.

[5] Markus Kuhn. *The TrustNo1 Cryptoprocessor Concept*, Technical Report, Purdue University, 1997-04-30.

[6] Microsoft. Next-generation secure computing base, 2003.

[7] Trusted Computing Platform Alliance. Trusted Platform Module, 2003.

[8] TPM Design Principles, Version 1.2. Trusted Platform Module, October 2003.

[9] Aucsmith, David. "Tamper Resistant Software: An Implementation"*Proceedings of the First International Workshop on Information Hiding*, 1996.

[10] Christian Collberg and Clack Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection, IEEE Transactions on Software Engineering, Vol. 28, Number 8, 2002.

TABLE II

MEMORY HIERARCHY SIMULATION PARAMETERS

Param	Alpha 21264 [17]	Intel XSCALE 80200 [18]
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

TABLE III

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

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

[12] E. Fredkin and T. Toffoli. *Conservative Logic*, In International Journal of Theoretical Physics, 21(3/4), April 1982.

[13] R. Bennett and R. Landauer. *Fundamental Physical Limits of Computation*, Scientific American, pages 48-58, July 1985.

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

[15] Andre DeHon. *DPGA-coupled microprocessor: Commodity ICs for the early 21st centry*, In Proc. of IEEE workshop on FPGAs for Custom Computing Machines, pages 31-39, April 1994.

[16] HMAC. Internet RFC 2104, February 1997

[17] Zarka Cvetanovic and R. E. Kessler.*Performance analysis of the Alpha 21264-Based Compaq ES40 System*, In Proc. of ISCA, pages 192-202, 2000.

[18] Intel 80200 Processor based on Intel XSCALE Microarchitecture Datasheet, Intel, January 2003.

[19] Specbench. Spec 2000 Benchmarks

[20] Doug Burger and Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0. Computer Sciences Department Technical report #1342*. University of Wisconsin-Madison. June 1997.