

TIVA: Trusted Integrity Verification Architecture

Mahadevan Gomathisankaran and Akhilesh Tyagi

Electrical and Computer Engineering,
Iowa State University,
Ames, IA 50011
{gmdev, tyagi}@iastate.edu

Abstract. We are moving towards the era of pervasive computing. The embedded computing devices are everywhere and they need to interact in many insecure ways. Verifying the integrity of the software running on these devices in such a scenario is an interesting and difficult problem. The problem is simplified if the verifying entity has access to the original binary image. However, the verifier itself may not be trusted with the intellectual property built into the software. Hence an acceptable and practical solution would not reveal the intellectual property (IP) of the verified software, and yet must verify its integrity. We propose one such novel solution, TIVA, in this paper.

1 Introduction

We are entering the era of pervasive computing where embedded devices have penetrated most spheres of human activity. These embedded devices carry a wide range of data ranging from sensitive personal information to military confidential information. Moreover, these devices need to interact frequently with the insecure world. Hence it is imperative to check frequently whether any malicious tampering of the software on these devices has occurred.

The different scenarios where such verification is beneficial, for example, are as follows.

- The field officer would like to ensure that her GPS has not been tampered with before entering the enemy territory. Note that the tampering adversary here is the GPS device. The military needs to distribute the binary image of the GPS software to the verifier so that the field officer can use the verifier to ascertain the integrity of the GPS software. The military, however, would be increasing the risk of compromising the IP of the GPS software by distributing the binary image to the verifier. Note that the IP adversary is the verifier (and not the device, which is a tampering adversary). The problem then is devising verification engine (verifier) architecture to minimize the risk of exposing the IP of the distributed GPS software.
- An executive would like to ensure that the software and/or data on her PDA has not been tampered with. She could have a verifier installed on her laptop to verify the PDA. There exists a conflict of interest between the software vendor and the PDA user. PDA user (or the laptop version of the verifier) requires the binary image of the PDA software for verification. The software vendor may be at the risk of compromising the IP of her software by distributing it to the PDA owner. Thus the verification architecture should safeguard both the party's interests.

- An organization would like to ensure that their routers were not tampered with. This case is pretty similar to the earlier one except that the verification would be performed remotely. The verification architecture should be robust enough to support the remote verification of the systems.

All these scenarios demand IP protection in addition to the mere verification of the software. The existing solutions like *SWATT*([1]) and *Genuinity*([2]) do not address the concern of IP protection and are very restricted to a certain class of devices hence not generally applicable.

Reverse engineering the low level code into a high level programming language is usually the first step in determining the embedded IP of a software. Such reverse engineering can lead to software piracy. Reverse engineering requires disassembling and decompilation of the instruction sequence. Static obfuscation techniques address this issue by hiding the instantiated instruction sequence. These obfuscation techniques embed the correction points for the control flow (the correct instruction sequence) in the image itself. Such instruction sequence obfuscation, however, applies only to the static program image. The instantiated instruction sequence is exposed during an execution.

In the case of verification model, the verifier needs the binary image for verification purpose only and not for execution purpose. In other words, an instantiated control flow path order is not important to the verifier. The verifier mostly needs only the memory address-content correspondence. Thus any obfuscation technique which modifies the static sequence of instructions need not embed the image with correction points. Such an obfuscated image becomes extremely hard to reverse engineer without the execution address sequence. In TIVA we use a permutation function to generate such an obfuscated image in order to provide the IP protection.

TIVA uses challenge-response protocol between the verifier and the embedded device. In order to keep the tampering adversary, the device, honest in its responses, the challenge has to be different (unique) for each verification. TIVA uses a unique permutation function for each verification to calculate a unique checksum or hash. The novelty of TIVA lies in the fact that it can achieve both IP protection and challenge-uniqueness through the use of a permutation function. TIVA uses a trusted hardware element in the embedded device to achieve this. But this trusted hardware is different from TCG or secure processors as it has very minimal hardware overhead.

The main contributions of this paper, TIVA, are

- identifying the need for IP protection for any practical integrity verification model for embedded devices
- providing both IP protection and challenge-uniqueness to every verification instantiation through permutation functions
- a reconfigurable circuit to achieve these permutation functions

The rest of this paper is organized as follows. Section 2 describes the problem and the assumptions under which the solution is valid. Section 3 explains the proposed solution. Section 4 explains the reconfigurable permutation unit which forms the basis to our solution. In Section 5 we put together these elements and explain the overall verification architecture. Section 6 discusses strengths and weaknesses of our proposed verification architecture. Section 8 concludes the paper.

2 The Problem

Integrity verification allows the verifier to assert that the binary image, which includes both code as well as static data, is as expected. Let \mathcal{E} be the device whose binary image needs to be verified, \mathcal{V} be the entity which would like verify the integrity and \mathcal{D} be the entity which distributes the software image I . The interactions between the entities are as follows. Recall that \mathcal{E} is the tampering adversary for the verification. However \mathcal{V} is the DRM adversary against whom we need to protect the software IP. Note that \mathcal{V} is a logical entity that can be physically realized either as a hardware or software unit separate from \mathcal{E} or it could be physically integrated as a software process or hardware unit within \mathcal{E} . In the later case, the hardware version of \mathcal{V} would have to be secured against observation and tampering from \mathcal{E} . The software process version would have to be obfuscated and hidden within \mathcal{E} along the lines of software watermarking [3] with unique secret handles for instantiating the verification and for observing the outcome.

- *Distribution*: Software vendor \mathcal{D} distributes the image I to verifier \mathcal{V} to verify the integrity of the corresponding software in the device \mathcal{E} .
- *IP Protection*: \mathcal{D} trusts the device \mathcal{E} to have sufficient protection mechanism to protect the IP of image I . Note that \mathcal{E} is protecting the IP of I against possible reverse engineering by \mathcal{V} . However, a direct distribution of I to verifier \mathcal{V} by software vendor \mathcal{D} increases the risk of IP compromise. \mathcal{V} could have simulation/emulation environment or use other mechanisms to reverse-engineer I . To avoid such a scenario \mathcal{D} would like to ensure that IP of the binary image I is protected despite its distribution to \mathcal{V} .
- *Verification*: \mathcal{V} would like to verify the integrity of the binary image I resident in the device \mathcal{E} . The verification process should be challenge-response based, i.e. the verifier \mathcal{V} should be able to generate a challenge at random, and based on the response from \mathcal{E} should be able to assert the integrity of the image I . The verification process should be robust enough so that it is able to detect replay and spoofing attacks.

The problem boils down to \mathcal{V} verifying the image of \mathcal{E} with respect to I without revealing its IP under the condition that \mathcal{E} is not tampered with in hardware. The binary image refers to both the code as well as static data.

3 The Solution

The three dimensions of the problem as explained in Section 2 are *distribution*, *IP protection*, and *verification*. We first present the solution to the problems of verification and IP protection. The distribution problem arises out of this solution.

A straightforward solution to the problem of verification would be to distribute the binary image I to the verifier \mathcal{V} . Hence the verifier can read contents of the \mathcal{E} and compare it against the received image. But there are several problems with this simple and seemingly perfect scheme.

First of all the requirement of IP protection is violated by this scheme, as the verifier \mathcal{V} could very well be an attacker who would like to reverse engineer the IP of the

image I . Another problem with this scheme is that it is highly inefficient. It will take time proportional to $N * c$, where c is the number of cycles required to read the memory content from the device \mathcal{E} and N is the size of the memory.

Earlier solutions like, *Genuinity* [2] and *SWATT* [1], addressed this problem by having a verification module in the device \mathcal{E} . This verification module receives the challenge and provides a response to the verifier \mathcal{V} . This verification logic is critically dependent on the following two dimensions.

1. binary image residing in the memory, I .
2. time to perform the verification, \mathcal{T} .

In such a verification module architecture, one solution would be to distribute the hash of the binary image to the verifier \mathcal{V} , and to ask the verification logic in the device \mathcal{E} to generate the hash as well, followed by a comparison of the two hashes. Any modifications in the binary image I will modify the hash and any modification to the verification logic to misrepresent the hash itself will result in a perceptible change to \mathcal{T} . Since only the hash is available to \mathcal{V} , no binary image I is provided, the IP protection problem is moot. But the drawback of such an approach is that the hash used in the verification is fixed. Any malicious software running in \mathcal{E} could spoof the verifier by responding with the fixed hash without having to recompute. The time to perform verification could be easily spoofed by the use of timers.

An alternative would be to request the verification module in the device \mathcal{E} to compute the hash of a variable subset of the image I . Since verifier \mathcal{V} can specify the subset at random the response to every challenge has to be uniquely calculated to thwart the replay attack. Similar method is used in AOL [18] and AIM [19]. In this schema though the verifier \mathcal{V} needs to be able to calculate the correct hash for more or less any subset of I (every challenge). It requires the entire binary image I for this ability. But this violates the IP protection requirement of our problem statement.

Yet another solution is to use keyed hash. The verifier \mathcal{V} can generate a random key and request the device \mathcal{E} to generate the hash for that key. This could also avoid the replay attack since the hash value depends on the key and the key is generated at random by the verifier \mathcal{V} . But this model also violates the IP protection requirement since the verifier \mathcal{V} requires the image I in order to calculate the hash for a randomly generated key. Another drawback especially applicable to a software based remote verifier is the ease of mimicking the device behavior. An impostor device \mathcal{E}' could replace \mathcal{E} such that both are behaviorally equivalent (say a malicious router). Moreover, \mathcal{E}' could be computationally much more powerful than \mathcal{E} , able to easily calculate the hash within \mathcal{T} from the unmodified original image. In reality, though, \mathcal{E}' could be executing a modified malicious image. Since there is no shared secret between the verifier \mathcal{V} and the device \mathcal{E} , any impostor could generate the correct hash, since the hash algorithm, key and the image are all known to the impostor. This idea was also used by Umesh et al. [4] to attack *Genuinity* [2].

Thus the solution to the verification problem is to find an *irreversible* hash or checksum function which generates a unique hash \mathcal{H} for every verification. This function should be such that within the given time \mathcal{T} the only way to generate \mathcal{H} is to execute the given verification function. Also this function should share a secret with the verifier. Thus if \mathcal{E} returns the required \mathcal{H} within the specified time \mathcal{T} then it verifies the integrity

of the device \mathcal{E} as well as the image in \mathcal{E} . Thus heart of the solution is in defining the irreversible hash function \mathcal{F} which generates \mathcal{H} . This \mathcal{F} and \mathcal{T} together constitute the signature of \mathcal{E} which is verified against the precomputed values by \mathcal{V} . This is the core of our proposed approach to integrity verification.

The required and desirable properties of the irreversible hash generation function \mathcal{F} are as follows.

1. It should be very fast and efficient. Hence any change in \mathcal{F} or its simulated/emulated version should result in a perceptible and observable change in the response time \mathcal{T} .
2. It should depend on the image I as well as on the challenge from the verifier \mathcal{V} . Thus for two distinct challenges, it should generate distinct hash values.

Algorithm 1. Irreversible hash function (Pseudocode)

```

for  $l = 0$  to  $N - 1$  do
     $hash = hash + (MEM[l] \oplus \pi(l))$ 
end for

```

Algorithm-1 shows such an irreversible hash function. This hash function calculates the checksum of the image I XOR-ed with permutation function π . $MEM[l]$ refers to memory contents of the image at location l . π refers to the permutation function which takes in a value from $0 \cdots N - 1$ and returns a value from $0 \cdots N - 1$. There are $N!$ possible distinct permutation functions. Verifier \mathcal{V} chooses a particular permutation function through the challenge. Device \mathcal{E} should use that specific permutation function while calculating the checksum.

The notable characteristic of the hash function shown in Algorithm-1 is that it uses the permutation function π to create the dependency between checksum calculation and verifier's challenge. In contrast, SWATT [1] used pseudo-random generator and Genunity [1] used architectural side-effects to introduce such dependency. The main reason behind our choice of permutation function is the additional capability of IP protection offered by these permutation functions.

Reverse engineering is the first step in determining IP of the software. In order to reverse engineer the control flow graph (CFG) of the image has to be reconstituted. This is done by disassembling and decompilation of the binary image. Various static obfuscation techniques ([22], [23], [24], [5]) try to achieve IP protection by either obscuring the disassembling stage or decompilation stage. But these techniques are limited by the fact that the statically obfuscated image should retain the same CFG as its original.

The degree of obfuscation required in our problem is significantly weaker. The verifier \mathcal{V} needs the image only for verification or to establish address by address correspondence of the contents of \mathcal{V} 's and \mathcal{E} 's images. The binary image held by \mathcal{V} is not executed. This weakens the obfuscation constraints as follows. Any static obfuscation applied to the binary image I distributed to \mathcal{V} need not retain the original CFG. Any permutation of the sequence of the bytes in the binary image I would obfuscate the CFG, in turn making the reverse engineering extremely difficult. Thus obfuscated image I_{obf} , which is a permuted version of the image I could be distributed to the

verifier \mathcal{V} without compromising its IP. Section 6.1 discusses in detail the strength of obfuscation function realized by permutation.

Our solution to the integrity verification problem which combines the permutation function to generate I_{obf} and the permutation function to generate hash to form a unified solution is as follows.

1. For every $(\mathcal{V}, \mathcal{E})$, \mathcal{D} generates a permutation function π_d and gives $(\pi_d(I), \mathcal{T})$ to \mathcal{V} .
2. \mathcal{D} secretly embeds π_d in \mathcal{E} .
3. For every verification, \mathcal{V} generates π_v and finds $\mathcal{F}(\pi_v(\pi_d(I)))$. It then gives π_v as a challenge to \mathcal{E} .
4. \mathcal{E} generates hash using π_v and π_d and reports it back to \mathcal{V} .
5. \mathcal{V} measures the response time \mathcal{T} .
6. \mathcal{V} can verify this *signature* with the precomputed one.

Figure 1 shows an example calculation of checksum by both \mathcal{V} and \mathcal{E} . In this figure obfuscated image I_{obf} is generated as follows. Let M_{obf} be the memory content of I_{obf} and M be the memory content of image I . Then $M_{obf}[\pi_d(i)] = M[i]$ for every i from 1 to N , where N is the size of the image. Note that image I is not necessarily limited to only instructions. The presence of static data could also obscure the disassembly which makes reconstruction of CFG more difficult. In this figure, verifier \mathcal{V} has the obfuscated image I_{obf} and device \mathcal{E} has the actual image I . \mathcal{V} generates π_v and calculates hash \mathcal{H} using Algorithm-1. Device \mathcal{E} uses the composite permutation function $\pi_v(\pi_d)$ and the actual image I to calculate the same hash \mathcal{H} .

A permutation function π with N values is $N!$ strong, which is slightly higher than 2^N by Sterling’s approximation of a factorial. Hence by choosing sufficiently large N we can reduce the probability of success through a *brute – force* attack. By choosing a different permutation function for every verification we avoid the *replay* attack. Attack by impostor is avoided as \mathcal{V} and \mathcal{E} share the permutation function π_d as the secret. Hence

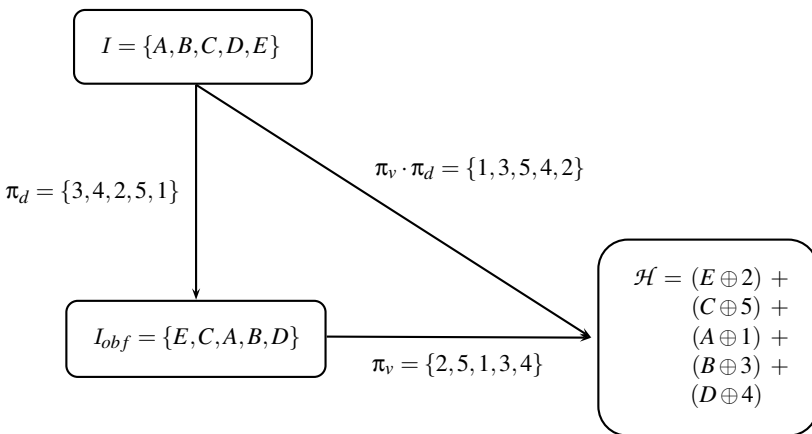


Fig. 1. An example hash or checksum function \mathcal{F}

the impostor needs to know π_d to generate \mathcal{H} . We have assumed that \mathcal{E} is protected enough not to reveal its stored secrets.

The distribution of the software image now involves four operations, namely, distributing image I to the device \mathcal{E} , generating the permutation function π_d , generating the obfuscated image I_{obf} and distributing it to the verifier \mathcal{V} . Various existing solutions are applicable to this problem. In the case of embedded devices it is most likely that the device vendor distributes the image as well. Hence the device vendor can maintain the association of π_d with the device's unique ID. Whenever the device is purchased or obtained by the verifier the vendor can generate the obfuscated image using π_d and distribute it with the device. Whenever the device needs to be updated with newer version of the image I the device vendor has to generate the corresponding I_{obf} and distribute it to the verifier \mathcal{V} . In the following section, we describe the circuit to realize the reconfigurable permutation function. This logic needs to be embedded into \mathcal{E} .

4 Reconfigurable Permutation Function Unit (RPU)

This unit is responsible for realizing the permutation function π . There are $2^n!$ permutation functions possible for a n bit input. Reconfigurable logic is well-suited to generate a large dynamically variable subset of these functions. Figure 2 shows one such schema for permutation of 10 address bits, but note that this methodology is extensible to any number of bits. Before explaining the blocks of Figure 2, we observe that there are $(2^{2^n})^n$ 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 ([6], [8]) with LUTs.

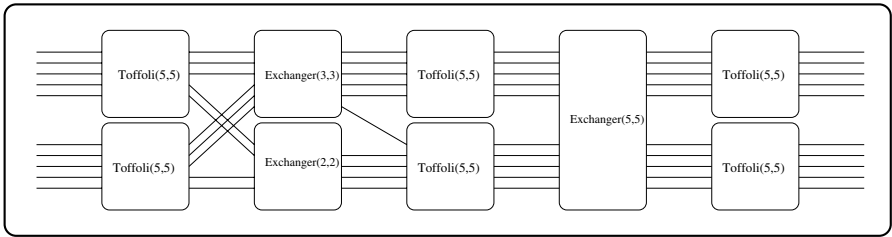


Fig. 2. Reconfigurable Permutation Unit (RPU)

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 [6] and Toffoli [7] have defined classes of reversible gates.

Definition 1. *Toffoli gate, $Toffoli(n,n)(C,T)$, is defined over a support set $\{x_1, x_2, \dots, x_n\}$ as follows. Let the control set $C = \{x_{i_1}, x_{i_2}, \dots, x_{i_k}\}$ and the target set $T = \{x_j\}$ be such that $C \cap T = \emptyset$. The mapping is given by*

$$Toffoli(n,n)(C,T)[x_1,x_2,\dots,x_n] = [x_1,x_2,\dots,x_{j-1},z,x_{j+1},\dots,x_n]$$

where $z = x_j \oplus (x_{i_1} \wedge x_{i_2} \wedge \dots \wedge x_{i_k})$.

Definition 2. Fredkin gate, $Fredkin(n,n)(C,T)$, is defined over a support set $\{x_1,x_2,\dots,x_n\}$ as follows. Let the control set $C = \{x_{i_1},x_{i_2},\dots,x_{i_k}\}$ and the target set $T = \{x_j,x_l\}$ be such that $C \cap T = \emptyset$. The mapping is given by

$$Fredkin(n,n)(C,T)[x_1,x_2,\dots,x_n] = [x_1,x_2,\dots,x_{j-1},p,x_{j+1},\dots,q,\dots,x_n]$$

where $k = x_{i_1} \cdot x_{i_2} \cdot \dots \cdot x_{i_k}$, $p = (x_j \cdot \bar{k}) + (x_l \cdot k)$, 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 2. 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 2 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

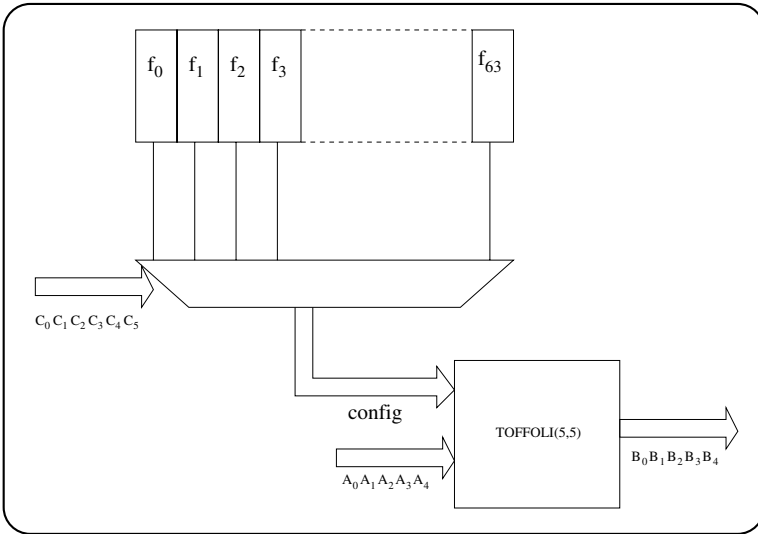


Fig. 3. Configuration Selection for each LUT

$S_{ok} = S_{ik}$ 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 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 [9]). In addition to 4 *input bits*, each LUT will also have 6 *configuration bits* to choose one of the 55 configurations (assuming some configurations are repeated to fill the 64 locations), as shown in Figure 3. 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 2.

4.1 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 address to the outgoing address). 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 permutation 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 2 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 1.

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 1 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

Table 1. Redundancy Estimation: # of Rndm Confgs = 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

reconfigurable permutation 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.

4.2 Area and Delay Estimation

Since we intend to use RPU in the embedded devices it should be both area and delay efficient. Figure 2 shows that RPU has $6 \times 5 - LUTs$ and 3 *shifters*. Each of these $5 \times 5 - LUTs$ takes 6 *configuration selection* bits and 5 *input bits*. Each *LUT* can be visualized as having a direct mapped cache with 64 sets and 32 bit cache line. Each cache line stores the *configuration bits* and one of which is chosen by the 6 *configuration selection* bits. One of these 32 *configuration bits* is chosen by the 5 *input bits*. Thus an *LUT* has a 256B direct mapped cache and a 32-to-1 multiplexer. Since all the 5 *LUT* use the same configuration selection bits we can group all these direct mapped caches and make it a single direct mapped cache with 64 sets and 20 byte cache lines. Figure 4 shows such a schema.

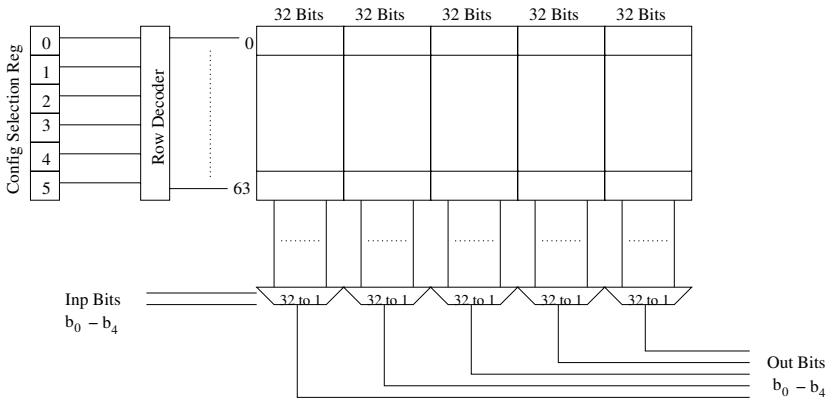


Fig. 4. A typical schema for 5x5-LUT

Table 2. Area estimate of RPU using CACTI [11]

Technology nm	Area mm ²
180	1.4526
130	0.7578
70	0.2196

Thus RPU has 6 1.25KB direct mapped caches. Since configuration selection bits will be preloaded, the delay incurred in accessing these caches would not have any impact on the access time of RPU. We used CACTI [11] to estimate the area requirement of RPU and Table 2 lists the area estimate for various process technologies. The other components of RPU are shifters and multiplexers. The shifters could be realized through 2-to-1 multiplexers. Since more than 99% of the transistors of RPU are contributed by the caches the area estimate of RPU could be equated to the area estimate of the caches.

To estimate the access time of RPU we should find the components which contribute to the access time. Since the configuration selection register will be preloaded the configuration bits will be available to the multiplexers. The access time can be given as,

$$T_{RPU} = 3 \times T_{32-to-1 MUX} + 2 \times T_{2-to-1 MUX}$$

We used HSPICE [10] to perform the delay estimation. We used pass transistor logic with appropriate drivers to design the multiplexers as they are area efficient. In order to optimize the delay of a 32-to-1 multiplexer we designed it as a 3-level multiplexer with first two levels being 4-to-1 multiplexers and the last one being 2-to-1 multiplexer. We used TSMC [14] and BPTM [15] models for the simulation. The results of the simulation are listed in Table 3. We will use these delay estimates while estimating the latency of this functional unit in the following section.

Table 3. Delay estimate of RPU using HSPICE [10]

Technology nm	Model	V_{cc} V	V_{th} V	$T_{32-to-1}$ ps	T_{2-to-1} ps	T_{RPU} ns
180	TSMC [14]	1.80	0.46	369	70	1.247
180	TSMC [14]	1.30	0.28	410	80	1.390
180	TSMC [14]	1.55	0.28	340	60	1.140
70	BPTM [15]	0.90	0.20	220	60	0.780

5 Integrity Verification Architecture

In Section 3 we outlined our basic solution for embedded device verification. In Section 4 we explained RPU which forms the basis of the proposed TIVA. In this section, we explain TIVA in more details. TIVA uses RPU, the hash function \mathcal{F} and the response time \mathcal{T} to provide the solution to the integrity verification problem.

5.1 XRPU

As explained in Section 3, the IP of image I is protected through π_d . In TIVA this is achieved by embedding this secret in the device \mathcal{E} . Hence \mathcal{E} should have a protected

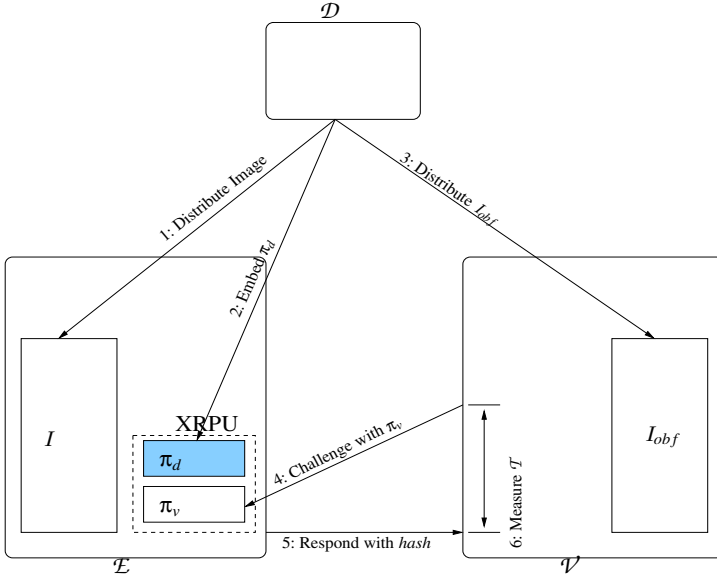


Fig. 5. Integrity Verification Architecture

hardware where this secret could be stored. From Section 4, we know that RPU has space to store all the 64 possible configurations for each LUT. Since π_d chooses only one of these configurations we do not need to store all of them. Thus \mathcal{E} should have a special RPU $_{\pi_d}$ which stores only the chosen configuration bits which amounts to 120 bytes for the LUTs and 3 bits for the exchangers.

\mathcal{E} should contain a second RPU $_{\pi_v}$ which is a generic one as explained in Section 4. This RPU is loaded with the configuration selection bits generated by \mathcal{V} . Since we want to protect the function π_d , we do not allow the input/output relation of RPU $_{\pi_d}$ to be visible. If π_d is allowed to be observed then I_{obf} could be de-obfuscated resulting in loss of its IP. Thus we create a single composite function unit XRPU, eXtended RPU, which contains both RPU $_{\pi_d}$ and RPU $_{\pi_v}$. It takes start address and configuration selection as input and produces the *hash* as the output. This XRPU generates all 1024 addresses sequentially from the start address and computes $hash = hash + \{MEM[addr] \oplus \pi_d(\pi_v(addr))\}$. This could be implemented as microcode or implemented in hardware. Since π_v is public its permutation function is known. Hence given $addr$ and $MEM[addr] \oplus \pi_d(\pi_v(addr))$ it is easy to derive π_d . Thus XRPU only provides *hash* as the output from which π_d cannot be obtained as it is an irreversible function.

5.2 Verification

As is the case with any encryption function, the algorithm of RPU is public. The secret is the *configurations bits*. Thus \mathcal{V} could be provided with a simulated version of RPU's algorithm or it could have a special application-specific hardware unit. To verify the authenticity of the image, \mathcal{V} generates the configuration bits for π_v randomly and

```

li    0,0    ; R0 counter
li    5,0    ; R5 LS word of checksum
li    6,0    ; R6 MS word of checksum
lwz   1,st   ; R1 starting address
L1:  add 1,0,1 ; add counter to address
xrpu  3,0    ; R3 = XRPU(R0)
lwz   2,0(1) ; load the content in R2
xor   3,2,3  ; R3 = R3 xor R2
srawi 4,3,31 ; R4 = sign bit of R3
addc  5,5,3  ; R5 = R5 + R3
adde  6,6,4  ; R6 = R6 + R4 + Carry
addi  0,0,1  ; R0 = R0 + 1
cmpwi 0,0,1024; is R0 < 1024
lt    L1     ; loop back

```

Fig. 6. An example PPC micro-code implementation of \mathcal{F}

computes the checksum as $sum = sum + (MEM[i] \oplus \pi_v(i))$. It then sends this π_v as a challenge to \mathcal{E} and measures the time of verification (response time).

Since XRPU is a hardware unit, the verification function \mathcal{F} , which we assume to be a microcode, could be very fast. As an example, in PPC the execution of one iteration of loop body for this function takes only 10 cycles assuming XRPU takes 2 cycles per operation. Example pseudocode is shown in Figure 6. This is very fast and efficient. Any small modification in the verification code results in perceptible change in the time \mathcal{T} of the verification process. Thus from the checksum and \mathcal{T} , \mathcal{V} can establish the integrity of the binary image in \mathcal{E} . Figure 5 explains various steps involved in the verification architecture.

5.3 Overhead Estimation

Since RPU_{π_d} stores only one set of configuration bits area of $XRPU \approx$ area of RPU , whereas $T_{XRPU} = 2 \times T_{RPU}$ as RPU_{π_d} and RPU_{π_v} are in series. Using the estimates from Section 4.2 we estimated the area overhead and latency of XRPU for various commercial embedded processors and the results are tabulated in Table 4. In summary, the area overhead of this scheme is fairly insignificant. We see that for all the processors the area overhead is less than 1%. Even for low end embedded processor with 10 mm^2 of area the overhead comes out to be 2.2% for 70 nm technology to 14.5% in 180 nm

Table 4. Latency and Area overhead estimation of XRPU

Processor	Technology	Package mm^2	Max Freq MHz	% Area Inc	Delay ns	Latency cyc
PXA 255 [12]	0.18 μ ,1.80V	17x17	400	0.50	3.367	2
PXA 26x [12]	0.18 μ ,1.30V	13x13	400	0.86	3.367	2
PXA 27x [12]	0.18 μ ,1.55V	13x13	624	0.86	3.147	2
PXA 800F [12]	0.13 μ ,1.20V	12x12	312	0.53		
PPC 750FX [13]	0.13 μ ,1.45V	21x21	900	0.17		
PPC 750Cxe [13]	0.18 μ ,1.80V	27x27	700	0.20	3.367	3

technology. The delay overhead is more easily hidden through pipelining. The overhead appears to be of the order of two cycles for most of these technology nodes, and hence fits nicely into any pipeline.

6 Discussion

6.1 Obfuscation Strength of Permutation Function (RPU)

In this section we quantify the *obfuscation strength* of the permutation function implemented using RPU. The aim of the permutation function is to obfuscate the instruction sequence. The first step in the process of reverse-engineering is disassembling the binary image. Once the instructions are disassembled their static sequence is used to reconstruct the control flow graph (CFG). Hence a measure of obfuscation could be derived from the dissimilarity between the original CFG and the CFG derived from the obfuscated static image.

The nodes in a CFG correspond to a basic block, a straight-line sequence of instructions. The permutation function rearranges the static instruction sequencing. This results in the modification of many basic blocks as constructed from the obfuscated/permuted image since there are no corresponding basic blocks in the original CFG. For instance, even if one instruction from an original CFG basic block is permuted away past a control instruction (a branch), a new basic block results in the obfuscated CFG. The edges in the permuted CFG similarly can either be completely new or may have a different source or target basic block. We will call a basic block or edge perturbed if there is no corresponding basic block (in the way of graph isomorphism accounting for new naming) or edge in the original CFG.

A permutation function that perturbs all the nodes (basic blocks) and edges from the original CFG achieves *complete* obfuscation. We define an analytical limited version of this notion that captures the similarities of the instruction sequences in the original image versus the permuted image. We will estimate what fraction of sequences of n instructions are preserved (or perturbed) from original to the permuted image for a large range of values for n . A typical basic block is 5 to 10 instructions long. Such a measure for $n = 5$ then estimates the fraction of perturbed basic blocks which constitutes a simplistic measure of obfuscation. We define such an obfuscation strength measure of size n , OS_n , for the permutation function as follows.

Definition 3

Let

$I \rightarrow$ Unobfuscated binary image

$I_{obf} \rightarrow$ Obfuscated binary image

$N \rightarrow$ Number of instructions in I ($|I| = |I_{obf}|$)

$S_n^j \rightarrow$ Sequence of instructions i_1, i_2, \dots, i_n in I
from j^{th} position where $1 \leq j \leq (N - n + 1)$

Then

$OS_n =$ % of S_n^j not in I_{obf}

Note that in our solution we permute the binary image in units of *words* (4 bytes). In some architectures (like x86) the instruction sizes are not fixed. Thus permutation could break some instructions giving rise to illegal or different instructions. Also the presence of static data in the image could cause the same effect. Hence this definition of *obfuscation strength* is very conservative and forms a lower bound.

To understand the definition of our *obfuscation strength* let us consider an example. Figure 7 shows an example permutation. In this example $|I| = |I_{obf}| = N = 5$ and S_2^j exist for $j = 1, 2, 3, 4$. In I_{obf} only S_2^1 exists unobfuscated. Hence $OS_2 = 75\%$ and OS_n for $n > 2$ is 100%.

As explained in Section 4 RPU has 39 *configuration selection* bits. It is highly improbable to find the *obfuscation strength* of RPU by exercising all the 2^{39} configurations. We generated 2^{20} random configurations and found the average *obfuscation strength* for various sequence sizes. We repeated this experiment several times to get a statistical validation of our experiment. All the values are listed in Table 5.

Table 5. Average *Obfuscation Strength* for 2^{20} runs

Seed	OS_5	OS_6	OS_7	OS_8	OS_9	OS_{10}	OS_{11}
0x031245f8	94.74	95.96	96.80	97.35	97.83	98.29	98.63
0x7fc5a2d5	94.75	95.98	96.82	97.37	97.84	98.30	98.64
0x015e8f8c	94.73	95.97	96.80	97.36	97.84	98.29	98.63
0x00231eea	94.74	95.97	96.80	97.36	97.83	98.29	98.63
0x0153d22e	94.75	95.97	96.81	97.36	97.84	98.30	98.64

We have listed the *obfuscation strength* for sequences of size from 5 to 11 as this happens to be the most frequent length for basic blocks. We see from Table 5 that with at least 95% probability our permutation function will obfuscate basic blocks with 5 or more instructions. This makes reverse-engineering of CFG from I_{obf} as difficult as the permutation function itself, which is $\approx 2^{34}$ strong.

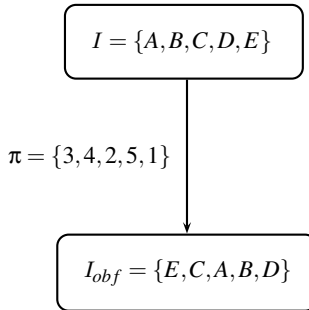


Fig. 7. An example Permutation

6.2 Attack Scenarios

A verification process using TIVA fails if one of following events occurs.

1. A malicious software executing on \mathcal{E} is able to generate the expected checksum \mathcal{H} in the expected response time \mathcal{T} .
2. An impostor system pretending to be \mathcal{E} , but with greater computational capabilities than \mathcal{E} , is able to generate the expected checksum \mathcal{H} in the expected response time \mathcal{T} .

For both these attacks to be successful the malicious software running on the device \mathcal{E} or the impostor system must know the composite permutation function $\pi_d \cdot \pi_v$. Since only microcode is able to exercise XRPU and it returns only the checksum value, it is not possible for the software running on the device to derive π_d . Thus the composite permutation function becomes as hard as the individual permutation functions which in our case is $\approx 2^{34}$ strong.

If an impostor system gets hold of I_{obj} then it is possible to generate the required hash without the knowledge of π_d . Only π_v and I_{obj} suffice. This attack could be avoided in two ways. The verifier \mathcal{V} could make sure that I_{obj} is stored securely and trust the device vendor \mathcal{D} to not release I_{obj} to anyone else. Another method is to extend the verification protocol in the application layer to add a unique ID to the device \mathcal{E} .

Secure storage of secrets in device \mathcal{E} is essential for the functioning of TIVA. Any attack that could reveal π_d would de-obfuscate I_{obj} thus compromising its IP. However, storing secrets in hardware is a well researched topic and solutions like battery-backed RAM as used in IBM's 4758 [26] secure coprocessor could be used. Ishai et al. [25] proposed circuit techniques to protect circuits against probing attacks. This could be used to store the secret within the chip resistant to probing attacks.

6.3 Flexibility of TIVA

We explained TIVA for 10-bit permutation functions in a 32-bit architecture producing a 64-bit checksum, thus handling a memory size of 4KB. But TIVA is not restricted to this memory size. For bigger memory sizes the verification function could be easily extended to produce 64 bits for every chunk of 4KB. As we mentioned earlier the checksum or hash generation function proposed in our solution is not ideal. It could be replaced with any other checksum generation function of any size. TIVA is not restricted to 32-bit architectures. It could very well be applied to 8-bit or 16-bit architectures. The microcode for the verification function could be modified accordingly.

TIVA is not restricted to Von Neumann or Harvard architectures either. As explained in [1] to verify Von Neumann architectures, in which program and data share the same memory, the device should be brought into some known state and then verification could be performed. This known state should be the one which is distributed as binary image to the verifier. Software vendors could distribute multiple such images for various checkpoints. In the case of Harvard architectures the program and data memory are separate hence it is sufficient to find the checksum of program memory alone.

7 Related Work

Seshadri et al. proposed software only attestation mechanism in SWATT ([1]). A software only solution will incur lower cost than an attestation technique that requires additional hardware. It can also be used on legacy systems. These were the two major selling points for SWATT. But SWATT is probabilistic, i.e. it accesses the memory based on a pseudo-random sequence. The verification procedure performs $O(n \log n)$ memory accesses, for memory size n , in order to access all of the memory with high probability. They generate 16-bit addresses from an 8-bit RC4 pseudo-random number by adding to it the current value of checksum. This could very well affect the probability distribution of the PRG sequence. The effect of this change on the probability of accessing every memory location in the system is not studied in the paper. Additionally, embedded devices in most cases are limited by battery power. Deployment of such a probabilistic method will incur energy penalty.

Kennell et al. proposed software only solution *Genuinity* ([2]) to address the problem of autonomous integrity verification of remote systems. This solution is applicable only to general purpose systems which expose architectural parameters like TLB miss counters, etc. They used these architectural side effects to uniquely generate a checksum through the verification procedure. They argued that this checksum cannot be generated whenever the verification procedure is modified or through other emulated/simulated systems. But Shankar et al. ([4]) proved that such a system based on architectural side effects is not sufficient to authenticate software.

Other solutions like IBM's IMA [16] use trusted hardware support [20] and require sophisticated OS support to verify the integrity. TPM provides root of trust for storage, for measurement, and for reporting. But TPM requires Public Key Infrastructure (PKI) and support of sophisticated message authentication algorithms like HMAC ([21]) to provide these trusts. The requirements of TPM and sophisticated OS support may be more than what an embedded device could offer. Moreover in IBM's IMA integrity verification is done only at the loading point. Hence any attack that occurs after the software is loaded will not be captured. Also the verifier is assumed to know the hash of the software or system configuration being verified. This again breaks our requirement of IP protection.

Thus earlier proposed solutions did not recognize IP protection as an important dimension in the problem of integrity verification. Our solution, TIVA, is different from these solutions in various aspects, such as

- TIVA uses a hardware component to aid the verification.
- TIVA is deterministic, i.e. it accesses each memory location at most once during verification.
- TIVA uses a shared secret between the embedded device and verifier to make simulating/emulating the device very difficult.
- TIVA uses permutation function to achieve both IP protection and randomness in hash generation function.

8 Conclusions

Embedded devices are omnipresent and pervade all facets of human life. This penetration is likely to only increase in the future. Their sheer numbers and wide presence make them amenable to tampering. A tampered sensor could misrepresent its environment (report no bio-hazard particles where there are some) or a tampered PDA could relay the private data of the user to a third party. Hence verification of these devices is a relevant problem. However, such verification needs to be extremely efficient and mostly automated given the sheer numbers of these devices. Moreover, the verification architecture will not be practical if it compromises the IP of the software running on these devices. This paper presents a novel hardware architecture TIVA and a schema for such a verification mechanism which satisfies all the requirements of a verification system without compromising the IP of the system being verified. We demonstrate that the silicon area overhead for TIVA is minimal, 1%, and its time overhead is completely absorbed in the pipeline.

Acknowledgments

Authors would like to acknowledge John Linwood Griffin and Ray Valdez of IBM TJ Watson Research Centre, Hawthorne for their valuable review comments.

References

1. Arvind Seshadri et. al. *SWATT: SoftWare-based ATTestation for Embedded Devices*, In Proceedings of ISSP 2004.
2. Rick Kennell and Leah H. Jamieson. *Establishing the Genuinity of Remote Computer Systems*, In Proceedings of 12th USENIX Security Symposium, 2003.
3. C. Collberg, and C. Thomborson. *Software watermarking: Models and dynamic embeddings*. POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages
4. Umesh Shankar, Monica Chew and J.D. Tygar. *Side effects are not sufficient to authenticate software*, In Proceedings of 13th USENIX Security Symposium, 2004.
5. Christian Collberg and Clack Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection, IEEE Transactions on Software Engineering, Vol. 28, Number 8, 2002.
6. E. Fredkin and T. Toffoli. *Conservative Logic*, In International Journal of Theoretical Physics, 21(3/4), April 1982.
7. T. Toffoli. *Reversible Computing*. Technical Report MIT/LCS/TM151/1980, MIT Laboratory for Computer Science, 1980.
8. R. Bennett and R. Landauer. *Fundamental Physical Limits of Computation*, Scientific American, pages 48-58, July 1985.
9. Andre DeHon. *DPGA-coupled microprocessor: Commodity ICs for the early 21st century*, In Proc. of IEEE workshop on FPGAs for Custom Computing Machines, pages 31-39, April 1994.
10. Star-HSPICE 2001.4 Avant! Corporation.
11. Steven J. E. Wilton and Norman P. Jouppi. *An Enhanced Access and Cycle Time model for On-Chip Caches*, WRL Research Technical Report 93/5, July 1994.

12. Intel PCA Processors Data Sheets, <http://www.intel.com/design/pca/applicationsprocessors/index.htm>.
13. IBM Power PC Data Sheets, <http://www-306.ibm.com/chips/techlib/techlib.nsf/products/>.
14. Taiwan Semiconductor Manufacturing Company Ltd, <http://www.tsmc.com>.
15. Berkeley Predictive Technology Model, <http://www-device.eecs.berkeley.edu>
16. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert v. Doorn. *Design and Implementation of a TCG-based Integrity Measurement Architecture*, In Proc. of the 13th USENIX Security Symposium, 2004.
17. Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert v. Doorn. *Attestation-based Policy Enforcement for Remote Access*, In Proc. of the 11th ACM Conference on Computer and Communications Security, pages 308-317, 2004.
18. AOL. The America Online Instant Messenger Application. <http://www.aol.com>.
19. PyxisSystemsTechnologies. AIM/oscar protocol specification: Section 3: Connection Management. <http://aimdoc.sourceforge.net/faim/protocol/section3.html>, 2002.
20. Trusted Computing Group, *Trusted Platform Module Specification*, Version 1.2, Revision 62, <http://www.trustedcomputinggroup.org>
21. HMAC. Internet RFC 2104, February 1997.
22. Cullen Linn and Saumya Debray. *Obfuscation of Executable Code to Improve Resistance to Static Disassembly*, In Proc. of 10th ACM Conference of Computer and Communications Security, pages 290-299, October 2003.
23. W. Cho, I. Lee, and S. Park. *Against Intelligent Tampering: Software tamper resistance by extended control flow obfuscation*, In Proc. of World Multiconference on Systems, Cybernetics, and Informatics, International Institute of Informatics and Systematics, 2001.
24. T. Ogsio, Y. Sakabe, M. Soshi, and A. Miyaji. *Software obfuscation on a theoretical basis and its implementation*, IEEE Transaction Fundamentals, E86(A)-1, January 2003.
25. Yuval Ishai, Amit Sahai, and David Wagner. *Private Circuits: Securing Hardware against Probing Attacks*, In Proc of CRYPTO 2003.
26. J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, and S. Weingart. *Building the IBM 4758 Secure Coprocessor*, IEEE Computer. 34: pages 57-66. October 2001.