# Towards the Issues in Architectural Support for Protection of Software Execution

Weidong Shi[†] Hsien-Hsin S. Lee[†‡] Chenghuai Lu[†] Mrinmoy Ghosh[‡]

College of Computing[†]
School of Electrical and Computer Engineering[‡]
Georgia Institute of Technology
Atlanta, GA 30332-0280
{shiw,lulu}@cc.gatech.edu[†] {leehs,mrinmoy}@ece.gatech.edu[‡]

## Abstract

*Recently, there is a growing interest in the research community to employ tamper-resistant processors for software protection. Many of these proposed systems rely on a specially tailored secure processor to prevent 1) illegal software duplication, 2) unauthorized software modification, and 3) unauthorized software reverse engineering. Most of these works primarily focus on the feasibility demonstration and design details rather than trying to elucidate many fundamental issues that are either "elusive" or "confusing" to the architecture researchers. Furthermore, many proposed systems have been built on assumptions whose security implications have not been well studied or understood. Instead of proposing yet another new secure architecture model, in this paper, we will try to answer some of these fundamental questions with respect to using hardware-based cryptography for protecting software execution. Those issues include, 1) Is hardware cryptography necessary? 2) Is per-process single cryptography key enough to provide the flexibility, inter-operability, and compatibility required by today's complex software system? 3) Is OTP (one-time-pad) in combination with "lazy" authentication secure enough to protect software confidentiality? 4) Is there way to protect software integrity using less hardware resource? Finally, the paper defines the difference between off-line and on-line attacks and presents a very low overhead security enhancement technique that can improve protection on software integrity over on-line attacks by several magnitudes.*

## Keywords

tamper resistance, security, copy protection,encryption, attack

## 1. INTRODUCTION

Recently, there is growing interest of designing secure processor architectures to provide a secure software execution environment on unprotected computing platforms. Such secure processor architecture supports tamper-resistance via new architectural and microarchitectural features. Coupled with hardware-based cryptography schemes, the secure processor architecture can be used to enable a secure environment where only authorized and un-tampered applications can be executed. The security is achieved by employing encryption/decryption and integrity checking mechanisms [7, 13, 12, 15, 16, 8] to protect data located in un-trusted external hardware devices, mainly the off-chip memory and hard disks. The fine grain on-demand integrity checking and decryption mean that only when the data and instructions are brought into the tamper-resistant processor will they be decrypted and their integrity be verified. When executing applications, the architecture ensures that sensitive data and instructions of the applications will not be disclosed at any point of time and the software integrity is always guaranteed as well. The secure processor architecture can be used not only to prevent possible *software attacks* on protected applications, but also to prevent *hardware attacks* which have not been addressed by other similar secure systems [2]. Due to the strong protections provided, the secure processor architecture is able to address many challenging security problems that have haunted computer industry for decades including software copy protection, anti-reverse engineer, virus protection, and trusted distributed computing.

Despite of the many published works on demonstrating designs of secure processor architecture, discussions and analysis on many fundamental issues of using hardware features to secure software execution are still lacking. This research deficiency could cause confusion and misunderstanding in the research community. For example, there is no published work in the literature that provides risk assessment on secure processor architecture, which is a typical research practice in the security community to study security schemes. In this paper, instead of presenting a new security architecture, we attempt to address some of the issues we have noticed that cause confusion and misunderstanding in the architecture research community for security. Our answers to these issues may seem controversial to some researchers. However, the purpose of this paper is not to present a final answer to these issues but to put them in focus to guide future work so that more secure and better systems can be developed.

The main contributions of this work are

- Presented the arguments for the necessities of hardware-based cryptography for protecting software execution through analysis of security requirements based on sample application scenarios.
- Presented the first time in the literature an attack model on lazy authentication based protection on software confidentiality using a detailed example.
- Detailed analysis of potential problems associated with using single per-process cryptographic key based approaches to protect today's complex software systems.
- Definition and discussion of the differences between *on-line attack* and *off-line attack* on hardware cryptogra-

phy. The taxonomy of *on-line/off-line attack*s is unique to the protection of software using hardware cryptography. This important property of software protection has been largely neglected in prior research works.

- Proposed a unique low cost *tamper prevention mechanism* (TPM) that can be used to strengthen protection on software integrity. Performance studies show that integrity protection based on 32-bit MAC (message authentication code) has substantial performance advantage over hash based integrity verification [1]. However, there is a concern about using "short" MAC to protect integrity. As presented in this paper, a 32-bit MAC when combined with the proposed tamper preventing mechanism can increase the difficulty of compromising integrity protection by several orders of magnitude. With a careful design, a 32-bit MAC based integrity protection plus the proposed tamper preventing technique can be more secure than a pure hash based protection.

The rest of the paper is organized as the follows. In the next section, we present scope on software security protection based on examining the security requirements of several application scenarios. In section 3, we briefly present some of the proposed protection systems and compare the differences. Then, in section 4, detailed discussion of several issues are presented where each issue is presented as a subsection. Section 5 concludes the paper.

## 2. SECURITY REQUIREMENTS FOR SOFTWARE EXECUTION

Depending on the types of applications, their operating environments, business model or even for political reasons, the requirements on software protection and the definition of security would be totally different. A system that is secure or a protection measure that is sufficient for one application could mean security disaster when applyed to a different type of application or used under a different business model. For example, some of the biggest security concerns for enterprise computing may be accountability, software virus, and access control. Due to the nature of enterprise computing, security is far more likely been compromised by software based attacks than by hardware-based tampering such as attacks involving a complicated logical analyzer. However, for consumer game console application, as indicated by history, hardware-based tampering has caused wide spread security breaches on consoles including compromising copy protection through user installed various types of cheating and spoofing devices.

To give more details, we present five different application scenarios ranging from military embedded system, game consoles, to distributed computing and examine their respective security requirements.

Firstly, high-tech military systems/weapons are increasingly dependent on complicated computer software. One of the many security concerns on high-tech military application systems is that they may fall into enemy's hand. If unprotected, the system along with its software can be studied by the enemy to come up counter measurement or counter system. Furthermore, the enemy can reverse engineer and design copied version of the system. Both are security nightmares that should be prevented regardless of the cost.

Secondly, software piracy has haunted software industry for decades. Many solutions have been proposed in the past to fight against software piracy. As indicated by the cases of XBOX security key breach [6] and compromise of Nokia N-Gage, achieving software copy protection on consumer platform is far more difficult than expected due to two specific attacks. One is hardware modification and the other is platform emulation. The first involves installing a spoof device such as Mod-Chip to break copy protection and the second bypasses security protection by running copied software through a software machine emulator (for example playing a PSX game on a PC).

Thirdly, most today's software systems often include program components coming from heterogenous sources such as device drivers provided by hardware vendors, OS provided by system developers, and software libraries provided by miscellaneous middle-ware developers and third party developers. Sometimes, developer of each software component may have its own requirements on security. For example, software drivers and BIOS often contain a great deal of information of the underlying hardware and architecture design. Device or platform developers may decide to hide these details from system developers, application developers, and software users. This could not only prevent competitors from studying the protected system but also prevent hackers from reverse engineering the driver/BIOS codes for writing a machine emulator. For middle-ware developers, they may want their libraries to be used by many application developers but at the same time, prevent application developers from knowing the underlying secret how their software works. Example middle-ware applications include expert systems, AI systems, financial analyzing kit, complicated control systems, application specific signal processing libraries and etc. The intellectual properties of these middle-ware systems often include complex software algorithms, data/parameters obtained through years of accumulated experimental study or observation.

Fourthly, privacy and secrecy of mobile software agents and mobile data has been intensively studied recently [14]. In many cases, the mobile software to be protected is not a stand-alone process, but a piece of program, called mobile agent. How to execute a piece of mobile code on a host machine without potentially exposing or disclosing both the software and its data is a great challenge.

Fifthly, internet based multi-player video gaming is growing rapidly. However, online multi-player gaming since the first day of its success has been mauled by rampant sometimes, wide spread "cheating"s [9]. Many of the cheating techniques involves reverse-engineer the client game software, modifying either the client code or data (so called authoritative clients) so that players using the hacked client will have advantages over others. The worst scenario is that the hacked clients or patches most time can be downloaded online, which clearly jeopardizes the entire business of online video gaming. How to prevent reverse engineer of the game clients and protect against tampering on the client game code and data is vital for this emerging market.

Most of the discussed security requirements can not be met by today's hardware design. Although researchers have tried to tackle some of the security requirements through software

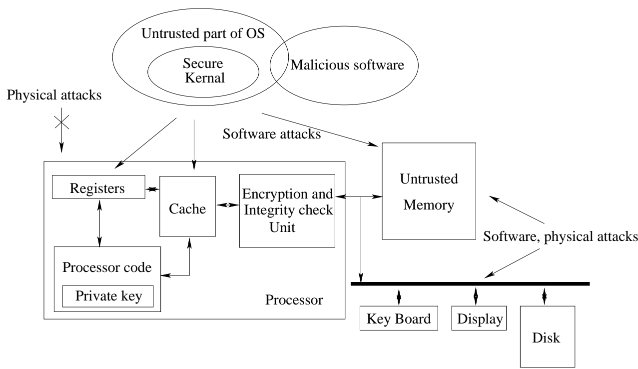


**Figure 1: Secure computing model**

based protection, the solutions are far from being satisfactory because there is always possibility of breaking a software based protection through either hardware based attack or software reverse engineering.

It is important to point out that the best practice of security hardware design is not to come up specific hardware features for each type of application, but to find out a set of basic security components behind the diversified security requirements. These basic security components can be implemented as trusted hardware security primitives. Various domain or application specific security requirements can be enforced through proper combination or usage of these trusted hardware security primitives. In short, protection of software security should include, 1) flexible, fine-grained protection on software confidentiality; 2) timely, rigorous checking on software integrity; 3) protection that prevents illegally duplicated software from being executed correctly.

# 3. RELATED WORK

A typical secure processor architecture model comprises of a tamper-resistant processor, external memory and peripherals as shown in Figure 1. Naturally, the protection boundary is drawn between the processor and external hardware units. Hardware units, like registers and on-chip caches, are protected from any possible attack while the remaining hardware units such as external memory and peripherals are considered vulnerable to physical attacks. Besides the aforementioned hardware, the secure computing model also includes a small trusted program, e.g., XVMM in *XOM* [7] and secure kernel in *AEGIS* [13]. The trusted program will be called secure kernel hereafter. The secure kernel has a higher privilege level than any other program including the regular operating systems and is responsible for performing encryptions/decryptions for the protected applications when their data are crossing the protection boundary. The secure kernel is also responsible for maintaining sensitive data that resides in private memory and registers during context switches. Note that confidentiality and integrity of "process context" are protected by the secure processor [1].

Some common features of proposed tamper resistant systems are described as follows.

- It is assumed that everything outside the CPU is unprotected and subject to malicious tampering. The physical RAM itself is neither protected and hackers could read/overwrite the memory content directly without involving the CPU. Furthermore, all the system/peripheral bus traffic is exposed and could be traced by the hackers.
- Like other tamper resistant systems, there is a pair of public-private keys associated with each secure processor. The secure processor's private key is permanently burnt inside the processor core and could not be accessed by software [7].
- Encryption/decryption and integrity check are supported by the hardware. When a data or instruction cache line is brought into the secure processor, it is decrypted and integrity of the entire virtual memory space is verified using a hash tree or MAC tree [1, 11]. When a cache line is evicted from the secure processor, it is encrypted and hash/MAC tree is updated. The keys used for encryption/decryption and integrity verification are set by the software vendors and encrypted by the secure processor's public key.

Most of the proposed systems support separate protection on software confidentiality and integrity. In XOM [7], a per-process encryption key (triple-DES) is used to decrypt software on-the-fly, while AEGIS [13] uses AES [5]. One major difference between Aegis and XOM is that Aegis employs an on-chip hash tree to verify integrity of the entire process space also in the execution time, thus preventing memory replay attack. As Yang et al. [15] indicates, block cipher based systems can incur substantial performance penalty. Systems using encryption schemes similar to one-time pad (OTP) and relaxed integrity check [12, 15] are proposed because they support faster software execution. Alternative solutions also aimed for better performance such as encrypting only small amount of carefully selected instructions, called software slices, can also be found in the literature [16]. In [10], a different architecture model, called MESA, is presented. Different from the previously proposed models that use a single cryptographic key to encrypt all the information in a process's memory space, MESA associates security attributes and cryptography keys with memory subspaces. Each protected memory subspace becomes an independent security domain, or security "sandbox", where information integrity and confidentiality of the memory subspace can be separately protected. One unique property of MESA is that it allows multiple protected "sandbox"es co-existing in the same memory space. This makes MESA a suitable solution for protecting shared libraries, mobile codes, device driver module, and etc. Table 1 lists some of the systems and their differences.

# 4. SOME ISSUES OF HARDWARE-BASED CRYPTOGRAPHY FOR SOFTWARE SECURITY

Although hardware cryptography is a promising direction for enforcing software security, there are still many remaining issues that have to be solved before it is mature enough

[1]Here process context refers to all per-process information that need confidentiality or integrity protection including but not limited to register values, page table, dirty cache lines, MAC/hash tree nodes, and etc. All those information are encrypted by the secure processor using secret key unique to each process. The root node of hash/MAC [1, 11] tree is preserved in securely sealed persistent memory during context switch.

Table 1: Some Tamper Resistant/Copy Protection System Comparison

| System | Runtime Integrity | Confi-dentiality | Cipher Cipher | Protection Range | Granularity |
|---|---|---|---|---|---|
| XOM [7] | none | yes | triple-DES | code & data | process based |
| AEGIS [13] | hash tree | yes | AES | code & data | process based |
| Yang et al. [15] | none | yes | DES OTP | code & data | process based |
| LogHash [12] (lazy authentication) | log hash | yes | AES OTP | code & data | process based |
| Zhang & Gupta [16] | none | yes | unknown | instruction slices | process based |
| MESA [10] | yes | yes | OTP | code & data | fine-grained, memory subspaces |

for real use. These issues include, testing issues, compatibility issues, programming model issues, privacy issues, performance issues, inter-operability issues, and security issues, etc. It is not possible to address all these issues in one paper. In this paper, we focus on some unique security and inter-operability issues of hardware cryptography. Although seemingly unrelated, they are important problems that a consensus on these issues if can be reached will definitely be beneficial to the future design of hardware cryptography based tamper resistant system. The four chosen issues are, 1) Is hardware cryptography necessary for building a tamper resistant system? 2) Is per-process single cryptography key enough to provide the flexibility, inter-operability, and compatibility required by today's complex software system? 3) Is OTP (one-time-pad) in combination with "lazy" authentication secure enough to protect software confidentiality? 4) Is there way to protect software integrity using less hardware resource?

## 4.1 Necessity of hardware-based cryptography

Does hardware cryptography provide more value than systems with only a trusted nucleus (secure kernel) and secure boot [3]? This concern on necessity of hardware cryptography is caused by the confusion of definition of security. It is true that hardware cryptography may not be absolutely necessary for some application scenarios. But as addressed in section 2, the diversity of security requirements for many important software protection scenarios demands on time, fast, secure protection on software confidentiality that can not be achieved without hardware supported cryptography.

Wherever there is software protection measure, there will be attacks. As history indicates, computer hackers/crackers tend to be well-knowledged, highly motivated, and sometimes well supported financially to crack a software protection system. Computer hackers often have many techniques, either in hardware and/or software, at their disposal to crack out the secret. Their efforts and dedication should never be underestimated. The instance of XBOX security key breach is one such example [6]. For copy-protection and software confidentiality, the problem becomes even harder. The entire protections on software confidentiality can be considered as broken if one adversary successfully reverse engineers a single copy of the protected software. Among the hacker's arsenal, two with demonstrated power of breaking protected software system are mod-chip based spoof attack and reverse engineer based emulator attack. Both attacks can be used to compromise copy protected software that does not use hardware cryptographic protection.

- **Mod-chip spoof attack.** Here we use the word mod-chip to refer to all the spoofing devices designed for bypassing or unravellig software protection mechanism. Mod-chips are low cost PCB attached to a platform designed for this purpose. Powerful and sophisticated Mod-chips can be used to record and replay memory and bus transactions. They can also be designed for hijacking another device's signal or launching device spoof attacks.
- **Reverse engineer based machine emulator.** Software copy protection is deemed broken if the protected software can be executed on a machine emulator without authorization. It is very difficult to fight against this attack without using software encryption. As suggested by history, machine emulator can be developed through reverse engineering BIOS and driver codes. Encryption of drivers and BIOS software can increase the difficulty of having emulator developed through simple reverse engineer. However, machine emulator can be alternatively developed through other means. In that case, software right can be protected by encrypting the application itself.

Please note that the XBOX security incident was caused by only one or two amateur crackers. When come to reverse engineer high-tech software systems for national security reasons, the cost, expertise, and resources would not be a concern at all.

## 4.2 Security of OTP and "lazy" authentication for protecting software confidentiality

To prevent hardware-based tamper, timely, rigorous protection on software integrity must be used. By "lazy" integrity checking, we mean that either integrity of instructions is not verified promptly or as frequently as on a per-instruction basis or architectural state can be altered before per-instruction integrity checking is completed. "Lazy" integrity check also refers to the situation that unauthenticated data is used as operant and the result is allowed to modify processor state before integrity of the source operant is verified. "Lazy" integrity check has been proposed for its better performance over rigorous and timely integrity checking mechanisms.

### *4.2.1 Program confidentiality*

Many nowadays'processors such as Alpha, MIPS, and ARM adopt RISC design philosophy. The simplicity of RISC instruction set enables more aggressive instruction fetching/decoding, pipelining and scheduling. However, on the other hand, the regular format and simplicity of RISC instructions also make it easy for adversaries to unravel encrypted RISC instructions. Here we use Alpha instruction set as an example

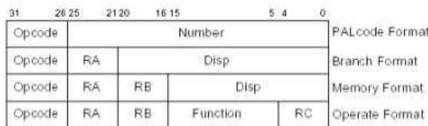


**Figure 2: Secure computing model**

to illustrate how an adversary can exploit the regularity of Alpha instruction set to crack instructions encrypted using one-time pad (OTP) generated from a per-process key. The attack assumes that the adversary can obtain front side bus traces of program execution via hardware based attack.

The vulnerability of RISC instruction set includes, 1) all the instructions have the same length and in many cases they are short, 16 bits, 24 bit or 32 bits. One weakness of short instructions is that it may be vulnerable to brute-force attacks; 2) the instructions are well formatted for easy decoding, for example, fixed opcode field. In the case of Alpha instruction, bit[31:26] is fixed as opcode. As shown in the example below, the biggest risk posed by this property of RISC instructions is that it may allow incremental guess of instructions. In such an attack, an adversary can divide each instruction into portions (opcode, operand one, operand two, and etc.) and launch brute-force guess piece by piece on each portion of the targeted instruction. This may significantly reduce the search space of brute-force attack. For instance, to apply brute-force attack on a 32-bit instruction, there are $2^{32}$ possibilities. However if the instruction could be divided into four 8-bit portions and each portion could be attacked in brute-force way, there are only 4*256 total possibilities, which is way smaller than $2^{32}$; 3) RISC philosophy advocates a small set of instructions and does not favor large number of complex instructions. This reduces the search space of possible instructions. Figure 2 shows some of the Alpha instruction formats used in the example attack [4].

To make it easier to understand, assume that the targeted Alpha binary does not use any dynamically linked libraries and all the instructions are packed into one code section and each instruction is encrypted using one-time-pad (regardless how it is generated). Also assume that the adversary has no prior knowledge of the program but is able to obtain front side bus traces of program execution. Further assume that the secure processor only performs a "lazy" integrity checking on the executed instructions.

In order to launch the attack, the adversary has to start from something s/he already *knew*. One candidate would be invariant instruction sequence that is pretty much fixed in almost every executable image generated by a compiler. For example, almost all the benchmarks in the compiled SPEC2000 binary have the same startup prologue instructions as the follows,

*sp,-16(sp)*
*stq zero,8(sp)*
*...*

Using these two known instructions as starting point, the adversary could launch known-plaintext attack and crack out more instructions that s/he could not guess so easily. A few good candidates would be the instructions in the middle of the code section. Assisted with the two known instructions, the adversary could control the next executed instruction by modifying the known instruction into a jump instruction with any target address s/he likes. We use sample code from SPEC2000's crafty as an example (Table 2), The code section has 37789 instructions. The list shows one candidate instruction (addq) in the middle that the adversary may choose as the jump target. The reason to choose instructions in the middle is because as shown later, they, after being altered into jump instructions by changing the opcode, are more likely to jump into valid code space than instructions close to the boundaries. Next, the adversary may perform a brute-force attack on the opcode of the targeted instruction by changing it. Since the Alpha instruction always uses bit[31:26] as opcode, the adversary could figure out bit[31:26] of the one-time-pad with at most 64 trials of opcode guessing. The opcode of the targeted instruction "addq" is 0x10. Assume that the adversary's first guess is opcode 0x4. The speculated bit[31:26] of the one-time-pad would be 0x21. Then s/he could change the instruction into an unconditional jump by altering bit[31:26] of the targeted instruction to the result of 0x21 $\oplus$ 0x30, where 0x30 is opcode of jump. Because the opcode guess is wrong, the altered instruction will be decrypted into an AND (0x11) instruction instead of a jump. Since there is no jump in the trace of instruction fetch, the adversary is certain that the guessed opcode is incorrect. Assume that the next opcode guess is 0x10 and is correct. This time, the program trace will show jump of program execution to target address 0x1200263E0 from address 0x12001139c( 0x12001139c + 0x5411*4). This will reveal bit[20:0] of the encrypted target instruction. The rest 5 bits (bit[25:21]) could be guessed by trying to alter the targeted instruction into a FETCH instruction, where all the 5 bits should be zero for a valid FETCH instruction. At most another 32 trials are required. Note that the above opcode attack can be launched in parallel using multiple machines with each machine taking one alternative guess. Given a moderate size of 64 machines, only under two parallel trials, the adversary is able to crack the encrypted target instruction.

The above case represents an ideal situation where the altered instruction jumps to an address within the code section. Since the displacement field of jump instruction has 21 bits, it is very likely that the targeted address may be outside the range of the code section. The adversary could tackle this problem in two ways, 1) modify the virtual address to physical address translation table so that the targeted address would be translated and fetched. Rigorous integrity checking and protection on TLB and process context of address translation may prevent such an attack; 2) Brute-force attack on both the opcode and the remaining displacement field bits whose range is outside the code section. In the above benchmark example, since there are about 37000 instructions, only the high 6 bits of the total 21 bits of the displacement field need brute-force guessing. Given 64 machines with each machine taking one guess of the opcode, at most 64 parallel trials are sufficient to break both the opcode and the remaining high bits of displacement field given the assumption that there is no alternative way for the adversary to tamper the address translation.

**Table 2: List of Crafty Code Section**

| Address | Plaintext | Ciphertext | Instruction |
|---|---|---|---|
| 0x120008840 | 0x23defff0 | 0x3127d04a | lda sp,-16(sp) |
| 0x120008844 | 0xb7fe0008 | 0x4c0d4ef4 | stq zero,8(sp) |
| ... | | | |
| 0x120010194 | 0x46520413 | 0xa0481bf0 | mov a2,a3 |
| ... | | | |
| 0x12001139c | 0x40c05411 | 0x9426814a | addq t5,0x2,a1 |
| ... | | | |
| 0x12002d670 | 0x23de0010 | 0x3704e241 | lda sp,16(sp) |
| 0x12002d674 | 0x6bfa8001 | 0x7a3250bf | ret |

Many embedded systems do not support virtual memory. For those systems, the tampered addresses could be observed directly on the bus. Since fetching the next instruction could be started before execution of the previous instruction is completed, even stronger instruction authentication is required. In such systems, jump targets of conditional and unconditional branches should not be fetched before integrity of the jump instructions are verified.

An adversary may use the above technique to figure out all the instructions. Alternatively, if only program code is encrypted, the adversary can use the following "short-cut" procedure. Firstly, s/he can figure out a short sequence of instructions (about 40) in the middle of the code section using method described above. Then s/he can speed up the attack by trying to alter other encrypted instructions into a STORE instruction. Take the `mov` instruction in Table 2 as one example. Through brute-force attack on the opcode, the `mov` instruction could be altered into,

*stw a12,a12(1043)*

To crack out the remaining 26 bits, the adversary may firstly transfer execution to the short code sequence which s/he has figured out, load a constant value to all the 32 Alpha registers by altering the cracked 40 instructions so that the computed data address (address register value + displacement) would be certainly within the space of data virtual address translation, then s/he can transfer execution to the altered targeted instruction. All these can be completed using less than 40 altered instructions. By observing the traces of data access, s/he would be able to figure out the last 16 bits of the targeted instruction (0x0413). This requires only one parallel trial or 64 single trials. To figure out bit[25:16], the adversary may repeat the same procedure. But instead of loading the same constant to all the alpha register, s/he will load a unique value to each Alpha register. Since the displacement is already known, subtracting displacement from the write address observed from the memory trace will reveal one unique value loaded to the alpha registers. This unique value will tell which alpha register is used and its register ID reveals 5 bit plaintext of bit[25:16]. The unique value stored will tell which register is used as data source and its register ID reveals the remaining 5 bits of bit[25:16]. In total, only two parallel trials are sufficient to crack the mov instruction.

As shown above, with only an amortized cost of two parallel trials/per instruction using 64 machines, the adversary is able to recover an OTP protected program within a reasonable time. Assume that it takes 30 seconds for the adversary to complete one parallel trial (in fact, this is an overestimate and the real time needed could be much less after the procedure is automated). It takes only about one and half month to crack out a program with about 64K instructions (256KB code size).

```
...
load r1, any address // load any data to r1
load r2, a chosen constant
if (r1<r2)
   goto address 1
else goto address 2
```

**Figure 3: Example Program**

We call the described attack technique, *"alter then trace attack"* (ATT attack). To use this attack, 1) the adversary must be able to alter a piece of software (program or data) bit-by-bit (satisfied by all the OTP based protection); 2) altered instructions can be executed and integrity of executed instructions or used data is not verified promptly or rigorously on per-instruction basis.

### 4.2.2 Data confidentiality

The above example shows how to break protection on program confidentiality when integrity of instructions is not verified promptly. Next, we will show protection on data confidentiality is also at risk. "Lazy" integrity check also refers to the situation that data source is used as operant and the result is allowed to change processor state before its integrity is authenticated. If altered instructions are allowed to be executed, an adversary can compromise confidentiality of any program data. Assume again that software confidentiality is protected using OTP and the adversary has successfully recovered a short program sequence (might use ATT attack ). Then s/he can convert the known short program sequence into an attack code sequence shown in Figure 3. The short code loads any data into the processor and compares the data with a chosen constant. If the secret data is 32-bit long, according to the principle of binary search, at most $\log_2(2^{32}) = 32$ trials are enough to recover the protected data. Alternatively, the adversary can treat data as instruction and use *"alter then trace attack"* to figure out its value.

If integrity of instructions is verified promptly but integrity of data is not, protection on data confidentiality may also be compromisable. It is hard to enumerate all the attack scenarios. Here we give one example, called *"link-list at-*

*tack"* to illustrate how to recover confidential data by only altering program data. Link-list is widely used in software program. One property of link-list is that the last node is always terminated with a NULL pointer. Assume that nodes of a link-list are protected using "lazy" authentication based OTP and the adversary knows where the link-list ends (the last node). Then, the NULL pointer becomes a known plaintext. Further assume that there is a secret data value x stored in memory location l, which the adversary wants to compromise. S/he can alter the NULL pointer into *l - node size + 4* so that the secret data becomes a node pointer. When the link-list is traversed, the program will try to use the secret data as a node pointer and issue a corresponding memory load which may reveal its value. Note that *"link-list attack"* is only one of the many possible attacks that can be tried for breaking data confidentiality. For example, *"string attack"* may be another candidate for compromising secret data when certain conditions are met. In *"string attack"*, assume that the software compares a string referenced by a string pointer with another constant string. If both the constant string and the string pointer can be modified, then the adversary can alter the pointer so that it points to some secret data s/he wants to compromise. Using method similar to the binary search attack example, s/he can recover value of the secret data.

In this section, we show some examples of breaking protection on software confidentiality under the situation that program or data integrity is not verified promptly. In short, all one-time pad (OTP) based approaches with "lazy" authentication check are potentially vulnerable. Approaches that check integrity in a timely fashion but have other flaws may also be vulnerable when certain conditions are met such as when the adversary is able to tamper the address translation.

## 4.3 Issue of single cryptography key

It seems that using a single cryptographic key to encrypt an entire process memory space is enough to provide security. However, in-depth study of single key based approach reveals that it is inefficient, inflexible, and sometimes, impractical for today's complex software system.

Firstly, in today's software system, it it hard to find applications that do not include pieces of software components come from heterogenous sources. Co-existing in the same memory space, these "external" software components could be static libraries provided by middle-ware vendors, dynamic libraries or other software modules that may not or may be also mapped to other tasks' memory spaces, mobile codes and data uploaded from other machines. Profiling of commercial Windows applications show that on average each application uses 20-30 shared libraries and in most cases, the combined code size of shared libraries is several times larger than the application itself. From the security perspective, under the single key approach, it is not possible for vendors of software components to enforce separate protection of their intellectual properties. For single key based approach to work, either all the external software components (often 70% to 90% of the total code size of an application under Windows System) left unprotected or have them duplicated and encrypted using the application's cryptographic key. The problems of duplicating shared libraries are four folds. 1) It is inefficient. Today's multi-task software system can easily have 60-100 tasks running concurrently. Each task can further have 20-30 shared libraries mapped to its memory space. Therefore, the memory overhead of duplicating shared libraries can be overwhelming; 2) It is not secure. If any application can have shared libraries being re-encrypted using its own key. It means no protection of the shared libraries themselves; 3) Encrypting shared libraries using application's cryptographic key can cause security problems for the application itself if the shared libraries contain malicious codes; 4) Simple duplication is not practical for libraries or software modules whose functionality is to achieve centralized management of software or hardware resources.

Secondly, it is very difficult if not impossible to enforce security on mobile codes under single key based approach. For mobile codes, the security requirements have two sides. On one hand, owners of the mobile codes/data may be concerned about potential disclosing of either the software or its data to the hosting system. On the other hand, the host system may be worried about potential security risk caused by malicious mobile codes. These two security goals may appear to counter each other because a perfect protection on mobile code may increase difficulty of protection of the host program because perfect protection on the mobile codes mean that the host has absolutely no knowledge what the mobile codes are doing. One advantage of MESA is that it allows multiple protected software execution "sandboxes" to exist in the same memory space thus can be used to provide security for both the mobile codes and the host program.

There has been a misconception that a secure kernel is hard to implement. In fact, under MESA, it is straightforward to implement a secure kernel as one protected "sandbox". Different from other "sandbox", the secure kernel's "sandbox" can be protected with hardwired secret cryptographic key that is known only by the secure kernel developers and the secure processor, therefore, preventing a different or maliciously altered secure kernel ever been loaded or executed by the system.

## 4.4 Protect integrity using less hardware resource

MAC (message authentication code) has been used to protect integrity of binary code and data in several proposed tamper-resistant systems. In some systems, a MAC is computed for each cache line size memory block of instructions or data and stored together with the memory block. When the memory block is fetched into the processor cache, integrity of the instructions or data is verified by re-computing a new MAC based on the fetched block and comparing it with the stored MAC. A mismatch between the two MACs indicates failed integrity check. A MAC tree can be also constructed to verify integrity of the whole virtual memory as shown in [1]. It is common knowledge that the length of the MAC itself is a direct measurement how secure the integrity protection is. A longer MAC often provides better protection against brute-force attacks than a shorter one. However, a long MAC also requires more hardware resources and incurs more memory overheads if they have to be cached inside the processor for performance reasons. Experimental results [1] show substantial performance advantage of using shorter MAC such as 32-bit MAC. The question is that, will a shorter MAC secure?

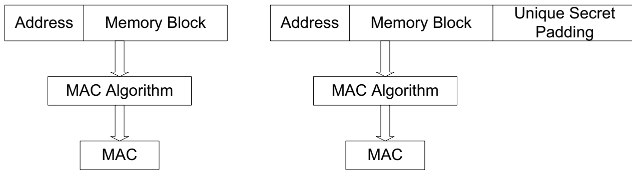


**Figure 4: On-line vs. Off-line Attacks On Integrity**

```
// Code Example Begin
push param1
push param2
push param3
push param4
push param5
call security_check
/*a jump to a subroutine */
tst ax, 0
/* assume return value in ax*/
bne security_failed
// Code Example End
```

**Figure 5: Example Code**

In this section, we present a simple hardware mechanism that can improve the strength of MAC by several magnitudes over *on-line attack*s on software integrity. There exists confusion in the architecture community on the difference between *on-line attack*s and *off-line attack*s on software integrity. *On-line attack* means that the attack has to be launched on the victim machine (targeted machine whose security an adversary wants to compromise) while in an *off-line attack*, presence of the victim machine is not necessary and the attack often can be conducted in parallel using multiple machines. Figure 4 shows two ways of computing MAC for each memory block. The first approach may suffer from *off-line attack*s on the MAC if the MAC algorithm does not require a key while the second would not because the secret padding is a secret unique to a machine and application thus preventing the attacker from computing a MAC without using the victim machine. Alternatively, if the MAC algorithm requires a cryptographic key unique to each platform and application, it will also prevent *off-line attack*s on the MAC. [2] To give a concrete example, considering the code sequence in Figure 5 and assuming an integrity code stored side by side with every 8 instructions,

Assume that the purpose of attack is to bypass the security check, there are two brute-force attacks the adversary could launch, one on the integrity code itself and the other one on the code sequence. First, brute-force attack on the integrity code, the adversary could alter the code sequence to another sequence (see Figure 6). This new code sequence very likely will have a different integrity code from the unaltered version. To fool the authentication check, the adversary could try to execute the altered code each time with a different random "integrity code" guess. If the integrity code is short, for example 16 bit long, after certain number of trials, a matching "integrity code" guess could be

[2]There exist *off-line attack*s on either cryptographic key or the secret padding itself. However, the key and padding can be long enough, say 256 bits long to make such *off-line attack*s impractical. Secret padding is encrypted using secure processor's public key when stored externally.

```
// Altered Code Example Begin
nop
nop
nop
nop
nop
nop
nop
nop
// Altered Code Example End
```

**Figure 6: Altering Both Program and Integrity Code**

```
// Altered Code Example Begin
mov ax, random_num
xor ax, ax
mov bx, random_num
xor bx, bx
mov cx, random_num
xor cx, cx
nop
nop
// Altered Code Example End
```

**Figure 7: Altering Program Only**

found. The adversary is able to test whether a trial is successful through program traces. If instruction fetch starts on the instruction after the last nop, the adversary knows that it is a success trial. However, this attack has to be conducted on the victim machine. Alternatively, the adversary could come up a huge number of "equivalent" attack code sequences and hope that one of them will have the same integrity code as the unmodified code sequence (see Figure 7). Different attacking code sequence will assign different random number to ax, bx, or cx. If the integrity code is short, by chance alone, some attack code sequence will have the same integrity code as the unmodified version. This allows the adversary to replace the original code sequence with a new one without change of the integrity code. However, this attack also requires that each altered code sequence be tried on the victim machine.

If a brute-force attack on integrity protection can be only launched online, a MAC as short as 32-bit can provide sufficient protection when combined with a tamper prevention technique presented next. To enhance security, a tamper prevention logic device, called *tamper prevention timer* (TPT) can be used to fight against online attacks on integrity code. TPT increases the difficulty of attack on integrity by deliberately inserting long time delay between integrity verification failures. Because the attack has to be launched on the victim machine, TPT can increase the amount of time required to break a system by several magnitudes, thus make a short MAC equally harder (in terms of required actual machine time) to break as a much longer MAC without using TPT. Pseudo-code of TPT is listed in Figure 8.

It is important to note that TPT is a hardware device embedded inside the processor core. It is not visible or accessible to any software. TPT has an output signal line called OK_line. During processor boot process, the OK_line will be

```
Assume
  tick_counter :
  penalty_register :
  failure_counter :
  failure_threshold :
  OK_line : output
When integrity check fails
  failure_counter++;
  if (failure_counter > failure_threshold)
   failure_counter = 0;
   tick_register = penalty_register;
   freeze processor;
   // execution can only be resumed through power cycling
  endif
For each core processor clock cycle
  if (tick_register > 0) tick_register–;
  if (tick_register==0)
   set OK_line;
  else
   clear OK_line;
  endif
```

**Figure 8: TPT Pseudo-codes**

checked. The processor will not start normal execution until the OK_line of TPT is set. Internal data such as data in tick_counter, failure_counter, and penalty_register are stored in persistent on-chip memory inside TPT. Their values can live across processor power cycling.

To give a concrete example of how TPT improves protection, assume that the delay for every 10 failed integrity verification is 1 minute. For a brute-force attack on a 32-bit MAC to succeed, on average $2^{31}$ number of trials are needed. This means 204 years.

It is important to point out that though TPT plus 32-bit MAC is reasonably secure against *on-line attack*s aimed to breaking integrity protection on a specific machine, a longer MAC such as 64-bit MAC might be preferred when an attack, we called, *"MAC collision attack"* is a security concern. *"MAC collision attack"* takes advantage of a well-known cryptography phenomena called the *birthday paradox* to reduce the MAC search effort comparing with a sheer brute-force search. The *birthday paradox* suggests that given some property (the birthday) that might have n distinct values and two set of values of the property, each $\sqrt[2]{n}$ values, there is a high probability that some value of the property in the first set is the same as some value of the property in the second set. To attack 32-bit MAC protected with TPT using *"MAC collision attack"*, the adversary must have access to $2^{16}$ ($\sqrt[2]{2^{32}}$) machines. Then s/he can come up $2^{16}$ random numbers as MAC guesses. Next, s/he can try the altered code sequence each time with a different MAC guess on the $2^{16}$ machines. According to the *birthday paradox*, there is a high probability that one of the machines will use one of the $2^{16}$ random numbers as integrity code of the altered code sequence. However, this very involved attack is not very practicable considering, 1) it requires large number of machines; 2) it does not make it easy to compromise any given machine. It may speed up breach of one machine among a huge number of machines with less number of trails. For most application scenarios, this attack is not a security concern at all. Alternatively, longer MAC such as 48-bit MAC or 64-bit MAC can be used together with TPT if *"MAC collision attack"* becomes a real security concern.

# 5. CONCLUSION

This paper presents an in-depth discussion of several issues of using hardware cryptography for protecting software confidentiality and integrity. The paper advocates the necessity of using hardware cryptography for preventing reverse engineering, and copy protection. It presents in detail why "lazy" authentication is not secure for protecting software confidentiality. Furthermore, it discusses many potential issues associated with applying single cryptographic key based approaches to protect software confidentiality in complex software system. Then, the paper defines the difference between *off-line* and *on-line attack*s and presented a security enhancement technique that can improve protection on software integrity over *on-line attack*s by several magnitudes.

# 6. REFERENCES


[1] M-TREE: A Fast Secure Architecture for Protecting the Integrity and Privacy of Software. *Submitted for publication.http: //www.cc.gatech.edu/people/home/lulu/Mtree.pdf*, 2004.

[2] The Trusted Computing Platform Alliance. https://www.trustedcomputinggroup.org/home. 2003.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65. IEEE Computer Society, 1997.

[4] Compaq Computer. *Alpha 21264 Microprocessor Hardware Reference Manual.*

[5] Federal Information Processing Standard Draft. Advanced Encryption Standard (AES). National Institute of Standards and Technology, 2001.

[6] A. Huang. Keeping secrets in hardware the microsoft xbox case study. *MIT AI Memo*, 2002.

[7] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh J. Mitchell, and M. Horowitz. Architectual support for copy and tamper resistant software. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[8] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 178–192. ACM Press, October, 2003.

[9] Matt Pritchard. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. http://www.gamasutra.com/features/20000724/pritchard01.htm

[10] Weidong Shi, Hsien-Hsin S. Lee, Chenghuai Lu, and Mrinmoy Ghosh. High Speed Memory Centric Protection on Software Execution Using One-Time-Pad Prediction. Report GIT-CERCS-04-27, Geogia Institute of Technology, Atlanta, GA, July 2004.

[11] E. Suh, B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, February 2003.

[12] E. G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings 0f the 36th Annual International Symposium on Microarchitecture*, December, 2003.

[13] E. G. Suh, D. Clarke, M. van Dijk, B. Gassend, and S.Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing . In *Proceedings of The Int'l Conference on Supercomputing*, 2003.

[14] T.Sander and C. Tschudin. Protecting mobile agents against malicious hosts. *Mobile Agents and Security. LNCS*, Feb, 1998.

[15] Jun Yang, Youtao Zhang, and Lan Gao. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December, 2003.

[16] Xiangyu Zhang and Rajiv Gupta. Hiding program slices for software security. In *Proceedings of the 2003 Internal Conference on Code Genration and Optimization*, pages 325–336, 2003.