Impact of Dynamic Binary Translators on Security

Yu-Yuan Chen[*], Youfeng Wu[+], Shiliang Hu[+] and Ruby B. Lee[*]
[*]Princeton University and [+]Intel

Abstract: *Dynamic Binary Translators (DBTs) allow programs written for a specific platform to be run on other platforms without the need for recompilation. They allow legacy software to be run on newer hardware architectures, they can perform dynamic optimization of software, and virtualization. Other benefits include providing enhanced security by dynamically adding checking code around possible software security vulnerabilities. However, before this is even considered, there are two aspects of DBTs that must first be addressed. First, are software protections provided by the application preserved under the runtime translation and optimizations done by a DBT? Will they be optimized out? We study a range of software protection techniques including Stackshield, Propolice and Stackguard, Libsafe, address space randomization, checksumming, watermarking, system call sandboxing, authenticated system calls, code obsfucation and morphing, anti-debugging, instruction-set randomization, and proof carrying code. Second, how is the DBT itself protected? How is its code cache protected? Without adequate protection, a DBT can be exploited by an attacker to cause disastrous system consequences. We propose three solutions. One solution adds a small set of hardware features to the microprocessor, as defined by the Secret Protection (SP) architecture, to protect the DBT and its code cache.*

1. Introduction

Dynamic Binary Translation (DBT) is a software technology that allows programs written for a specific platform to be run on other platforms without the need for recompilation. This not only introduces the opportunity for legacy software to be run on newer hardware architectures, but also enables dynamic optimization of software. The key capability of DBT is that it can transform and optimize any binary before it is executed on the processor, thus enabling many new features on existing binaries, such as virtualization [1], redundant execution for reliability [2], information flow tracking for security [3], dynamic voltage-frequency scaling for power management [4], etc. In this paper, we study StarDBT [5], an application-level DBT (running in user space), to see whether the translation and optimizations it performs affect the security of the original programs or not, and to examine ways of protecting the DBT itself.

When security protection is provided by software, and if this code undergoes standard DBT translation and optimizations, what happens to the security protections? Do optimizations done by a DBT preserve the security protections provided in the original code? Will they be optimized out? We study a range of software security protections to check if the translation and optimizations performed by a DBT affect the security provided to the original program.

A DBT's resources can be used to increase system security. For example, it could add checking code around potential software security vulnerabilities, such as locations where buffer overflow attacks might occur. A DBT could also add some randomization techniques (e.g., address or instruction set randomization) to thwart attacks. However, as the DBT is only another layer of software, if the attacker knows the existence of the DBT, it would not be difficult to attack the DBT itself and the DBT itself becomes a target.

Especially for an application-level DBT which runs in user space, we need to study how the DBT itself can be protected. How is its code cache protected? Without adequate protection, a DBT can be exploited by an attacker to cause serious system consequences. We discuss solutions based on moving the DBT to the system level, or alternatively using a small set of hardware features (as in the Secret Protection architecture **[6,7]** to protect the DBT and its code cache.

In section 2, we discuss different types of software security techniques that can be provided to an application program. In section 3, we describe our experimental setup and methodology for testing whether the security provided by these software security techniques is preserved under DBT translation and optimizations. In section 4, we provide the results of our tests and explain why the StarDBT we tested preserved the security properties provided, or not. We also suggest why other types of DBT might fail to preserve security properties, and also how to improve the DBT to not perturb security properties provided by application or system software. In section 5, we propose solutions for protecting the DBT, including a hardware technique that can protect the DBT itself, while leaving it in application space. In section 6, we present our conclusions and suggestions for future work.

2. Software Security Techniques

In order to see if a Dynamic Binary Translator will affect the security protection provided by different software products and techniques, we need to first understand what *types* of protection are provided and *how* they are provided. Hence, we illustrate the variety of software techniques that may be used to enhance security by preventing software attacks. These include preventing buffer overflow attacks, detecting hostile code modification attacks, checking system calls, mitigating reverse engineering, and checking security assertions. We describe a dozen different software techniques under these five types of protection. (Note that some of these techniques could be classified under more than one of these five categories.)

2.1. Preventing Buffer Overflow attacks

Perhaps the most common attack is the buffer overflow attack. There are many different software solutions for this, but it still remains one of the most prevalent software vulnerabilities in both legacy and new code. Buffer overflow attacks can cause return address corruption or function pointer corruption, leading to some forms of dynamic hostile code insertion or illegitimate library calls.

A buffer overflow attack occurs when data is written into a variable without checking if the length of this data exceeds the size of the variable. This can over-write other data, for example, the return address of a procedure call. This is illustrated in Figure 1, which shows a buffer overflow attack with ***return address corruption*** in the stack.

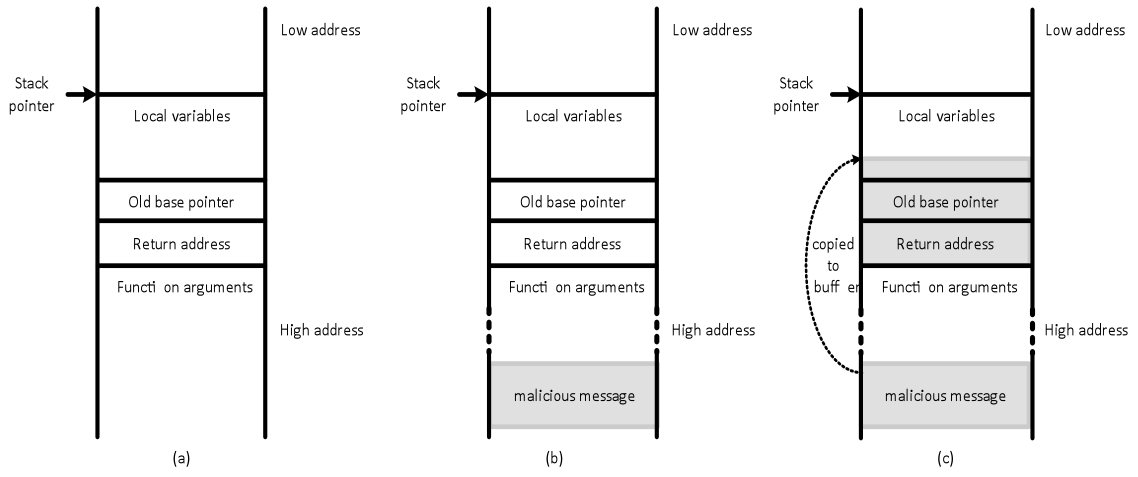

Figure 1: (a) a typical stack frame, (b) malicious message received from the network, (c) stack buffer overflow when message copied to local buffer without checking the size of the message.

By overwriting the return address, the attacker can alter the program control flow on a procedure return to jump to an address in the stack where hostile code has also been inserted by the buffer overflow. This hostile code, to which the program jumps, can gain root shell access. After this hostile code is executed, another jump instruction is executed to the original return address.

Alternately, a ***return to libc attack*** can also be launched by a buffer overflow attack, where the return address is changed to a legitimate library call entry-point. This causes a library call – that should not otherwise be called at that point in the program – before returning to the original return address. The return-to-libc attack is also based on overwriting return addresses on the stack; however, instead of transferring control to malicious inserted code, return-to-libc returns to existing functions in the system while overwriting the contents on the stack to feed bogus parameters for the existing functions to perform malicious operations for the attacker. Since return-to-libc does not require injecting new code into the system, even if the overflow buffer is not large enough for malicious shell code, it can still perform a successful attack. However, the attacker needs to figure out the address of the desired libc function such as `system()` or `printf()` to perform the attack.

a. Stackshield

Stackshield [8] is a software mechanism to prevent return address corruption by a buffer overflow attack. It is a gcc extension that maintains a (software) shadow stack for return addresses. It copies the return address to the beginning of the DATA segment, a location that cannot be overflowed, on function prologs, and checks if the two values of the return address are different on function epilogs.

b. Propolice and Stackguard

Propolice [9] is a patch to gcc. It inserts canaries to check for buffer overflow, as does StackGuard [10], to protect the return address on the stack. A canary is a bird used in mines to detect toxic air in the mine – the canary will die first from toxic air. Here, a "canary" is a random value inserted in the stack, such that any buffer overflow will change the value of the canary before damaging the return address; hence on a procedure return, the value of the canary is first checked to see to see if it has changed - if not changed, then the return address can be trusted. Additionally, Propolice reorders the declaration of local variables to place buffers before pointers to avoid the corruption of pointers that could be used to point to arbitrary memory locations.

c. Libsafe

Libsafe [11] intercepts known-vulnerable library calls and substitutes them with a safe version to ensure no that no buffer overflows can occur. This prevents buffer overflow vulnerabilities and attacks in these "safe" library calls. Libsafe does not require source code or recompilation and checks for violations at runtime. It is loaded as a linked library during runtime and checks a few "unsafe" functions, e.g. `strcpy(), strcat(), scanf(), memcpy()` etc. Libsafe intercepts these known-vulnerable library calls and substitutes them with a safe version to ensure no overflowing can occur.

d. Address space randomization

Address space randomization [12] is a technique devised to thwart buffer overflow attacks by introducing randomness into the memory locations of certain system components: stack, heap, BSS, etc. For example, the return-to-libc attack, discussed above, needs to know the virtual address of the libc function to be written into a function pointer or return address. If the base address of the memory segment containing libc is randomized, then the success rate of such an attack significantly decreases. Figure 2 (adapted from [12]) shows an example of the address space layout before and after randomization is applied. Linux application program code usually starts from 0x08048000, followed by the data segment that stores all initialized data and the BSS segment that stores uninitialized data. Dynamic Shared Objects (DSO) segments are where the shared libraries are placed.

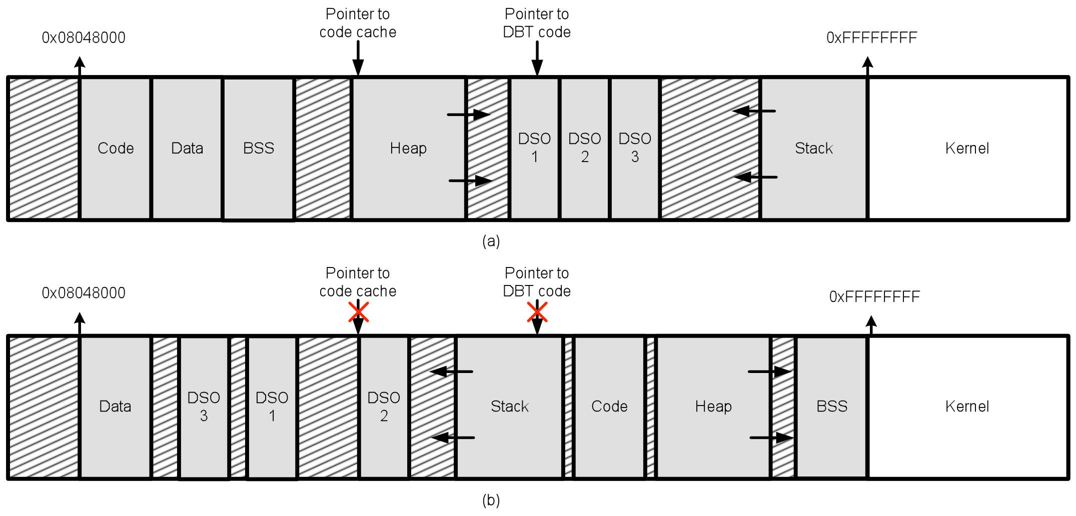

Figure 2: Address space randomization (a) typical address space layout. (b) layout after randomization.

For the StarDBT that we test, the StarDBT program is loaded as a dynamic library, residing in one of the DSO segments. The code cache of the application-level StarDBT is allocated in the user heap space.

2.2. Detecting hostile code modification attacks

Hostile code injection into a program, or code modification, can be detected by checking that the code has not changed from a known good version, We call this "static" code modification if it occurs when the program is not active, e.g., while the program was residing on disk. Checking that the code has not been maliciously modified is typically done only on program launch. Dynamic code modification occurs when the program is active – during runtime. The buffer overflow attacks, described earlier, enable dynamic hostile code insertion and modification.

The software Code Integrity Checking techniques described here are typically used to detect static code modification. However, they could potentially be adapted to detect some forms of dynamic code modification. We describe two different software techniques which add software to re-compute and check a checksum (or hash) on the code, or check for an inserted watermark.

e. Self-checksum

Self-checksum [13] is a technique where the program checksums itself to make sure the integrity of the protected part of the program is not compromised. A code producer would first compute a checksum (or hash) over the critical parts of the program that is to be protected, and insert that information into the program itself. In addition, the code producer inserts hash calculation and verification code throughout the program execution, to calculate the hash of the critical parts and compare with the inserted known "good" hash of those critical parts of the program. Therefore, any modification to the critical parts of the program will be detected.

Note that checksumming is not cryptographically strong, and can easily be defeated by an attacker, who can change the code then change the checksum value which is kept in the program itself. A better technique is to use a keyed-hash, or Message Authentication Code (MAC) rather than a checksum. Such a cryptographic hash function has strong collision resistance (unlike a checksum). Further, a keyed hash function (MAC) prevents an attacker from generating a new MAC value to correspond to the changes he makes in the code. This is because he does not have the key, which is kept securely elsewhere, and not with the program itself.

f. Watermarking

Watermarking [14] exploits the inherent redundancy in how instructions may be specified to encode a secret message in the program's binary. The exploitation is based on the fact that a given operation can be represented in many ways. For example, adding the value 50 to register `eax` could be represented as either "`add %eax, $50`" or "`sub %eax, &-50`". By replacing selected original machine instructions with functionally-equivalent instructions, we can encode a secret message into the program binary and later decode it to extract the message. The protection given by using this technique is that, if an attacker modified the disk image of the binary by inserting malicious code, the watermarked message would be destroyed and we would be able to detect that the binary had been modified. The watermarking technique works as follows: a checking program will first read in another program binary in which the watermark is going to be inserted, and change the instructions with functionally-equivalent instructions to encode a secret message. Afterwards, the checking program could read in the watermarked program's binary to decode the secret message. The watermarked program is not affected in terms of program execution result and the checking program only reads the watermarked program as static input.

2.3. Protecting System Calls

A common characteristic of many attacks is to exploit the system call interface to perform malicious actions. Incorrectly invoking a system call can do a great deal of damage to the system. For example, if a system call is invoked at the wrong time or in the wrong sequence, the system might be compromised. Forrest et al [15] confirmed that due to the different system call behavior when the system is attacked by the `sunsendmailcp` script [16], a local user may gain root access to the system. Another way of compromising a system call is by modifying system call parameters [17]. For example, the actions taken by a system call that normally retrieves the current user's file could be changed to access another user's file by including path traversal "../" characters as part of a filename request.

g. Sandboxing System calls

All sensitive system resources are usually accessed through the OS system call interface, e.g. file operations, network sockets, etc. System call sandboxing monitors all system calls made by an application by inserting checking instructions before and after the monitored system call and blocking the call if it violates some predefined security policy.

h. Authenticated system call

One technique proposed to protect system calls is the authenticated system call [18]. This transforms a system call to include extra arguments that specify the policy of that call and a MAC to guarantee the integrity of that policy. The policy specifies the allowed system call name, call site, control flow, and constant parameter values to be constrained for a system call. The policy is encoded into a string, and a Message Authentication Code (MAC), or keyed hash, is computed over the string with a cryptographic key that is available only to the kernel. This allows detection of any tampering of the string. The encoded string and its MAC are added to the system call as extra arguments. Before the kernel executes the system call, the MAC is re-calculated and verified to guarantee the integrity of the system call.

Before any program in the system is installed, it has to go through a trusted installer program, which extracts the allowed behavior for each system call by static analysis and then rewrites the binary to insert the policy and MAC. At runtime, each system call is intercepted and the MAC is checked to verify the integrity of the system call. If the behavior of the system call matches the policy, it is allowed; otherwise it is rejected and the process is terminated. Therefore, only trusted programs that have the authenticated system calls and pass the integrity check can execute on the system; regular programs cannot run on the system.

2.4. Mitigating reverse engineering

An attacker often first performs some reverse engineering to see if the software has exploitable security vulnerabilities, what and where they are, and where certain information is stored. Reverse engineering may also be performed to understand how secret or proprietary software works.

i. Code Obfuscation and morphing

Obfuscated code [19,20,21] is source code that has been transformed to make it harder to read and analyze, in order to prevent reverse-engineering of the code. Obfuscated code has the same function as normal code, just that the code is hard to understand without some processing. A simple code example of C that prints out prime numbers less than 100 is given in obfuscated form in Figure 3. Detailed directions on how to transform the original source code to obfuscated code are available, e.g., in [22].

```
_(__,___,____){___/__<=1?_(__,___+1,____):!(___%__)?_(__,___+1,0):___%__==___/
__&&!____?(printf("%d\t",___/__),_(__,___+1,0)):___%__>1&&___%__<___/__?_(__,1+
___,____+!(___/__%(___%__))):___<__*__?_(__,___+1,____):0;}main(){_(100,0,0);}
```

Figure 3: code obfuscation example for a C program

Some code obfuscation techniques simply add unrelated code into the binary. Other obfuscation techniques include the following:

- Lexical Transformations scramble the identifiers used in the programs, making it difficult to get semantic context from the identifiers, e.g., deposit (amount) is transformed to a (b).
- Control flow obfuscations [23] introduce redundant predicates whose outcome are known at the obfuscation time, but are difficult to deduce. The resilience of these obfuscations is directly related to the resilience of the opaque predicates on which they rely. Opaque predictors based on the intractability of static pointer analysis problem have been proposed.
- Data obfuscation [24] is achieved by modifying data structures to obscure their semantic meaning, e.g., variable splitting. Bool A -> int a1, a2. A = TRUE -> a1 = 0, a2 = 1, A = FALSE -> a1 = 1, a2 = 0.
- Inter-module call relation obfuscation [20] makes it a very complex problem to determine the address a function pointer points to, in the presence of arrays of function pointers.

Code morphing is code obfuscation done to the object code to deter disassembly and reverse engineering and analysis of object code. This technique breaks up the protected code into several processor commands and replaces them by others, while maintaining the original function. It turns binary code into an undecipherable mess that is not like normal compiled code. An example of code morphing is given in [25].

j. Anti-debugging techniques

Some hackers use debugging software to gain knowledge of how programs work, thereby allowing the hacker to find vulnerabilities that he/she can exploit. Therefore, commercial software often employ some anti-debugging techniques to prevent hackers from gaining this knowledge. There are various anti-debugging techniques, e.g. detecting SoftICE (a kernel-mode debugger for Windows), detecting VMware [26], searching for names installed by SoftICE in memory, etc. In general, these anti-debugging methods are simple instructions and do not depend on any dynamic behavior. However, one particular anti-debugging technique needs special attention: it checks execution timing to determine if being debugged.

```
xor ecx,ecx
rdtsc
add ecx,eax
…
rdtsc
sub eax,ecx
cmp eax,0FFF
jb short _FAST_ENOUGH
…
_FAST_ENOUGH: -> Code continues here
```

Figure 4: Timing-based anti-debugging detection

In Figure 4 the `rdtsc` instruction reads the time stamp counter and returns a 64-bit value in register `eax` that represents the count of ticks from processor reset. When being debugged, i.e. stepping through the program, the distance between the values in `eax` will be higher when the program is being debugged than when it is not being debugged. For emulators or interpreters, the overhead could be great enough to cause the program to always exit.

k. Instruction set randomizations

Randomizing the instruction set (i.e., the instruction encodings) [27] can deter remote attacks assuming that the attacker cannot obtain the encryption key that is used for randomizing. The idea is to randomize the entire instruction by XOR'ing the instruction with a key, as illustrated in Figure 5 The decoding (de-randomizing) to get the plaintext instruction from the encoded (randomized) instruction can be done in hardware or in software.

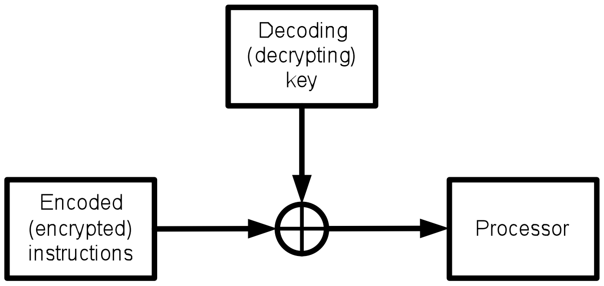

Figure 5: Decoding instructions in instruction set randomization (from [27])

Note that this form of instruction-set randomization is a very insecure form of One-Time Pad (OTP) encryption, since the encoding key is repeated over and over in the key pad of the OTP cipher. It would be rather easy for an attacker to break this encoding and discover the key. However, the purpose here is randomization to introduce diversity to thwart attacks and deter reverse engineering, rather than strong encryption.

2.5. Checking Security Assertions

l. Proof-Carrying Code

Proof-carrying code (PCC) [28] is code that carries an attached "safety proof" to prove that the code abides by certain security policies. The program binary is just a normal binary. The attached proof is written in Logical Framework (LF) and is encoded to be later checked by a "proof checker". All of the proof generation, encoding and proof checking are done statically—before the program is actually executed. Once the program passes the proof checker, the program will be executed as "trusted".

3. Experimental Setup and Methodology

We investigated several of these software protection techniques with and without using StarDBT. StarDBT is an application-level dynamic binary translator, and like most other sophisticated DBT systems [5, 29,30], it adopts the two-phase translation strategy. It uses a simple fast translator for cold code translation and once a workload hotspot is detected, it applies optimizations. The simple and fast translation tries to generate target code with minimal runtime overhead. Control transfer instructions such as branches, function calls and returns need to be rewritten. This involves some translation table lookup and dispatch code. The simply translated code can also be instrumented with profiling code to collect block frequency information to recognize program hotspots for optimizations.

For program hotspots, StarDBT forms straight-line traces (superblocks) to optimize, such as code layout and partial redundancy elimination. The translators place generated code in the code cache for later reuse.

The runtime dispatcher maintains a translation lookup table. The execution of the native IA32 code is achieved via the execution of the translated code and dispatching in the runtime system.

To verify that StarDBT does not compromise the security enhancement in the applications, we run the security-enhanced applications by themselves and also with StarDBT, under security attacks, such as the "attack benchmark" suite [31]. If the application, with and without DBT, can detect the same attacks, we consider that StarDBT does not compromise the security. Otherwise, we analyze the compromised security situation and determine how we may enhance StarDBT to prevent the situation from happening.

Since StarDBT is just a particular implementation of dynamic binary translation, some other types of DBT may compromise a security feature that is preserved by StarDBT, or vice versa. We identify this possibility by detailed analysis of the software security enhancement techniques (discussed in section 2), and point out the possible DBT translation and optimizations that may compromise the security provided.

4. Experimental results

We present the results of our experiments, described in the previous section, under three categories. Table 1 summarizes our overall findings. In section 4.1, we list the software protection techniques (DBT friendly techniques) that are not affected by DBT optimizations. In section 4.2, we list those software protections (DBT-tolerable techniques) that are preserved under certain conditions, and describe these conditions. In section 4.3, we list those software protections (DBT-troublesome techniques) that are not preserved by an application-level DBT, but can be preserved if the DBT is moved to the system level. In each case, we explain why the translation and optimizations done by the DBT we tested either preserved the software protections or not. We also discuss what might happen with other types of DBTs. We base our explanations on the following fundamental requirements for a DBT.

1) DBT must translate a program into a semantically equivalent version, in terms of all visible system states, including the detection of self-modifying code and retranslate them when necessary.
2) DBT must preserve the relative code and data layout of the original program in each of the data, bss, stack, and heap segments. This is required as some program may have built the relative offset in the program logic, although the runtime system is allowed to load each segment in a statically unknown location.
3) DBT must preserve self-referencing, i.e. the original program must be kept untouched and any self-reference to the original program during the translated execution must be faithfully provided.
4) DBT must preserve precise exceptions of the original program, i.e. all the executions must occur in the same order and the same places as in the original execution.

4.1 DBT-friendly techniques

For **Stackshield,** the shadow return stack value and the checking code are inserted statically. Their operations are part of the required behavior of the transformed program, and DBT will preserve the correct execution of these statically-inserted transformations. Specifically, the values in the shadow return stack cannot be determined as redundant by DBT if there is any code that may overrun the original stack, and therefore DBT will preserve the checking code, and will translate and execute it correctly.

Similarly, for **Propolice and Stackguard**, the special "canary" data value inserted onto the stack and checking code are inserted statically. If there is any chance of stack overrun, DBT cannot determine that the checking of the canary value is redundant computation, and thus is required to guarantee the correct execution of the input binary with its static Propolice or Stackguard additions.

Libsafe may substitute one library call with another dynamically at program load time. In this case, DBT will only translate the library calls after the substitution and thus a program under Libsafe will work correctly. If Libsafe substitutes library calls during program execution, DBT will detect the substitution as self-modifying code, and will still be able to translate and execute the newly substituted library code correctly.

For **address space randomization**, since the randomization is done at program loading time, DBT is required to *not change* the original memory layout within each segment, so the randomized address space will not be affected by DBT operations. For an application-level 32-bit DBT, the DBT needs to place its code cache in the same address space as the user application. There is a chance that the code cache may conflict with some of the user code/data. When this happens, DBT will relocate itself to another unused space and thus there will be no problem.

For **watermarking,** a secret message is embedded in the binary's disk image. Since the watermark checking program reads the image to decode the secret message and DBT translation will not modify the disk image at all, watermarking will not be affected by DBT.

Sandboxing system calls involves inserting checking instructions before and after the monitored system calls. Since the checking code is inserted for a valid purpose, DBT will not remove the checking code and so it will not compromise the protection.

For **Proof-Carrying Code**, if this proof-carrying code remains static, it will not be affected at all by DBT. Since all of the proof generation, encoding and proof checking are done statically -- before the program is actually executed, a DBT would not interfere with any step within the PCC framework.

Table 1: Summary of Preservation of Software Security Technique's protections under DBT

Software protection technique	Execution with DBT
Preventing Bufffer Overflow attacks	
a. Stackshield	OK
b. Propolice and Stackguard	OK
c. Libsafe	OK
d. Address space randomization	OK
Detecting Hostile Code Modification attacks	
e. Self-checksum (and self-MAC)	OK if self-referencing property guaranteed
f. Watermarking	OK
Protecting System Calls	
g. System call sandboxing	OK
h. Authenticated system call	This could be affected
Mitigating Reverse Engineering	
i. Code obfuscation and code morphing	May make the code less obfuscated
j. Anti-debugging	OK, except for some timing-based methods
k. Instruction set randomizations	DBT may not work with HW decryption of instructions; OK with SW decryption.
Checking Security Assertions	
l. Proof-carrying code	OK

4.2. DBT-tolerable techniques

Self-checksum should not be affected by DBT as the checksum is validated by the program itself and DBT ensures that all self-referencing refer to the original binary, not the translated code. However, for some DBTs that patch the input binary to link with the optimized code [32], the checksum may change because the patched code is different from the original code, so it will change the checksum. However, product quality DBT is required to support correct self-referencing at all times so this should not be a problem for a product quality DBT.

4.4. DBT-troublesome techniques

Authenticated system call replaces original system calls at program installation time with new system calls carrying additional parameters for authentication by the OS syscall checker. For binaries running under DBT, the system call is made from the code cache instead of the original code address. Furthermore, DBT may perform branch optimization which may result in a different control flow. If the policy string takes runtime call sites and control flow into consideration, this could be affected by DBT.

For **code obsfucation**, DBT nay affect some methods, especially the ones that simply add unrelated, redundant code into the binary. Since a DBT may perform sophisticated optimizations, it may optimize such code away and thus undo the obfuscation effect. The good news is that the recent code obfuscation techniques also try to make the code hard to optimize, and hence the obfuscation performed by the tools cannot be easily reversed by StarDBT.

In particular, the four types of obfuscations discussed in section 2.4 will not be compromised by DBT. Lexical transformations make it difficult to get semantic context from the identifiers -- but since this does not change the binary code, DBT will not affect it. Control flow obfuscations introduce redundant or opaque predicates -- but since these are based on the intractability of the static pointer analysis problem, DBT will not be able to compromise this. Data obfuscation is achieved by modifying data structures to obscure their semantic meaning -- but since DBT does not change data layout, it will not affect this transformation. (The variables are split statically in the example in section 2.4, and hence DBT will not impact this.) Inter-module call relation obfuscation makes it hard to determine the address a function pointer points to -- but since "point-to" analysis is difficult to perform at runtime by DBT, this technique will also not be affected by DBT.

For **code morphing**, DBT could, in theory, optimize the morphed code and revert it back to more readable binary instruction sequences. Therefore, this technique is not fundamentally secure under DBT code transformation that is not aware of the morphing. However, in practice; the binary codes are typically morphed in such a way that it is not easily optimized away by a fast runtime DBT optimizer.

For **anti-debugging**, since these methods consist of simple instructions and do not depend on any dynamic behavior, they should not be affected by DBT translation and optimizations. For anti-debugging techniques that check execution timing, it is possible that DBT may translate the code into a code sequence with a different timing. However, since the timing constraints for detecting a debugging event is usually quite large and DBT does not change the execution timing much, this technique should work with DBT. Note that timing-based anti-debugging techniques may result in false positives anyway -- if events like interrupts or context switching occur. Hence, DBT will not make this situation worse.

Instruction set randomization can deter remote attacks assuming that the attacker cannot obtain the encryption key that is used to randomize the instructions. Whether DBT will affect the security of this technique depends on how the randomization system is designed. If the system's decryption is done in hardware, the encrypted code will be fed into the decoder of the DBT, causing the DBT to fail completely, and the program will not work normally. The DBT can just leave the code un-translated and un-optimized – which may be OK in some situations, but not in cases where translation is necessary for correct execution of legacy code on a new machine. On the other hand, if the decryption is done in software, DBT can use the already decrypted instructions and the execution will not be affected. However, the DBT must be trusted to see the decrypted code, or even to have access to the secret key for decryption.

If a DBT is to co-exist with a system enhanced with instruction set randomization, both DBT and the decryption engine of the randomization should be incorporated into the hardware (or firmware) to be truly protected and, furthermore, to allow the DBT to use the decrypted machine code for its translation and optimizations.

4.6. Summary of Results

Although the 12 cases examined above are not exhaustive, they are representative and do include some of the most recent software protection techniques. With the results and discussion given in these cases, it seems that, as long as the implementation of the DBT is transparent and correct, most of the software protection mechanisms are preserved by DBT, except for a few cases where side-channel information or hardware is used for protection.

We make the following general observations based on our experimental results and analysis:

1) If the software protection is done statically, or at load time, then the DBT does not affect it – unless redundant information is added that the DBT can easily detect as redundant and may optimize away. (This is true for some simple, and probably less effective, types of code obsfucation and morphing.)
2) The software protections that operate at runtime should not be affected by DBTs if they are executed by the protected application program itself – this is due to the guarantee of correct self-referencing behavior that all product-quality DBTs should provide.
3) However, if the checking of the application program to be protected is done by a different program at runtime, based on the runtime behavior of the protected application code, then this can be problematic for the DBT since the dynamic code executed may be different from the application's static code. (This is the case for the time-dependent anti-debugging technique and for some types of authenticated system calls.)
4) If the instructions are transformed into a form not recognizable by the DBT, it would not work. Otherwise, the DBT has to be disabled, or it has to be trusted to see the decrypted instructions, or the key for decryption, before it can apply its translations and optimizations.

5. Protecting the DBT itself from attackers

An application-level (or user-level) DBT is as vulnerable as the application it protects, since it is just another level of application itself. It could potentially cause the application to be compromised without being detected if the DBT does not implement any protection to its code cache, or if the DBT's code itself is changed by an attacker. For example [33] reports a vulnerability with DBT via interprocess communications.

Alternative solutions to protecting the DBT's code are:

- Move the entire DBT to the kernel to get a system-level DBT
- Integrate DBT as a part of the hardware or firmware component
- Use Secret-Protection (SP) hardware mechanisms [6,7] to protect the DBT itself as a Trusted Software Module.

5.1. System-level DBT

The DBT can be integrated as part of the OS kernel, i.e., the DBT runs as a kernel module and the code cache is only written in kernel mode by the DBT module. The code cache can be mapped read-only and/or execute-only into user space. Thus, both the DBT module and code cache are protected by the OS kernel. However, if the kernel itself is not protected, then it can also be compromised by an attacker. Especially as commodity OS are getting increasingly larger and more complex, it is hard to guarantee that the OS is free from bugs and software security vulnerabilities that can be exploited by attackers.

Alternatively, if a trusted Virtual Machine Monitor (VMM) layer is present in the system, a system-level DBT can reside there. Since this VMM will run at the highest privilege level, we assume that the enhanced hierarchical ring protection mechanism present in the processor ISA (e.g., using VT) can be used to protect both the VMM and the DBT. Since VMM is typically much smaller than OS kernels, it should, in theory, be less vulnerable to attacks.

5.2. Integrate DBT as part of the hardware or firmware component

A DBT system can also be integrated as part of the processor or platform microarchitecture, such as the Transmeta code morphing software [34]. In this case the DBT is coded in internal implementation ISA and the translated code is placed in memory hidden inside the processor. Thus DBT will be as trustable as the processor itself. .

5.3. Use SP architecture to protect the DBT

As an application layer, DBT is as vulnerable to attacks as any application. We propose that a small set of hardware-software mechanisms, called the Secret Protection (SP) architecture [6,7], can be used to protect the DBT as a trusted software module (TSM).

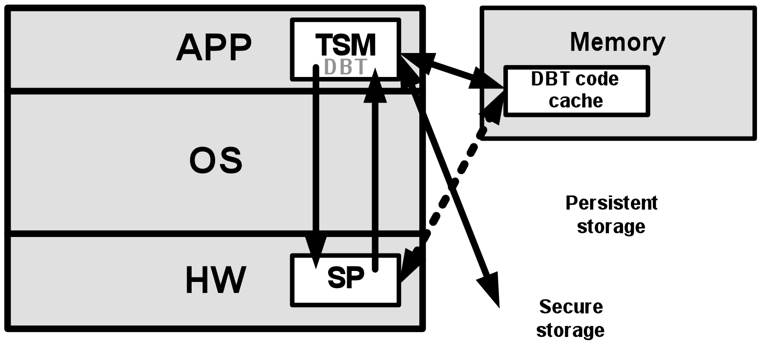

Figure 6: SP protection of DBT and code cache (white = the trusted parts.).

Figure 6 shows a conceptual view of the SP architecture consisting of a Trusted Software Module (TSM) in application space directly protected by hardware (SP) mechanisms in the microprocessor, even if the operating system has been compromised. With the size and complexity of commodity operating systems, this is not unlikely. SP architecture provides 3 main protections: (1) The TSM has access to pervasive secure storage. (2) SP hardware provides a secure execution environment for the TSM by protecting the intermediate data the TSM generates during its execution. (3) The TSM's code is integrity protected, so it cannot be changed by an attacker without detection. We describe each of these SP protections in the following paragraphs, after we first describe the SP hardware roots-of-trust added to the microprocessor.

SP contains two hardware roots-of-trust, a Device Root Key (DRK) and a Storage Root Hash (SRH), that protect the TSM and can only be accessed by the TSM (see Figure 7). Each SP device has a different DRK value (installed at deployment in its DRK register – not burned in at manufacture) that cannot be accessed by any software, except the TSM which can use an SP instruction, DRK_derive, to derive new keys for encryption or decryption from the DRK. No other computer can derive these keys without knowledge of the DRK.

The TSM has access to pervasive secure storage whose integrity is protected by an encrypted Merkle hash tree with the root of this tree stored in the SRH register in the microprocessor chip. The SRH can only be accessed by the TSM. Each level of the hash tree encrypts and hashes its sons using an encryption key and a hash key derived from the DRK. This pervasive storage structure can be stored in the untrusted disk, and no one, not even the OS, can access it except the TSM with the correct DRK. This pervasive secure storage is used by the TSM to store critical secrets, such as long-term public and private keys, certificates, licenses, encryption keys and security policies.

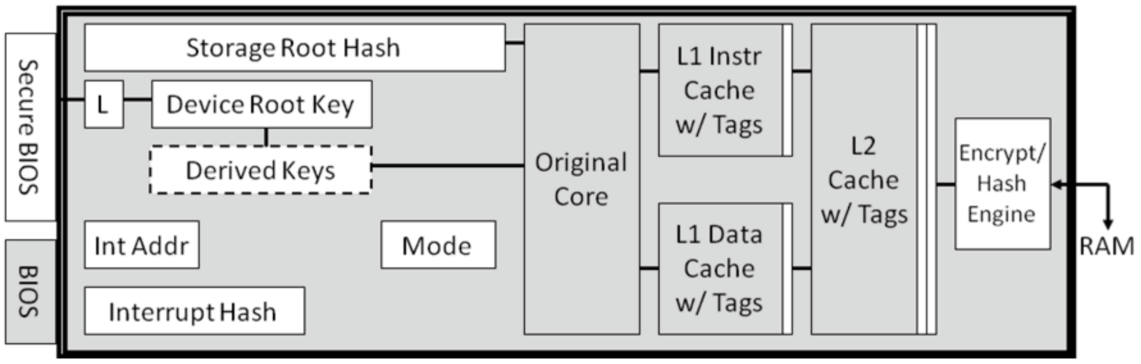

Figure 7: New SP components in microprocessor (white areas).

SP hardware provides a secure execution environment for the TSM so that no secret information accessed by the TSM is leaked out of the microprocessor chip. Since the security provided by SP depends so heavily on the DRK and SRH roots-of-trust, they must not be leaked out during TSM execution. Hence, SP provides a Concealed Execution Mode for TSM execution. SP considers only the microprocessor chip to be the physical security perimeter, i.e., only information inside the microprocessor chip is considered trusted and safe from physical probing and eavesdropping attacks – all information outside the microprocessor chip are considered untrusted, including information in the external main memory. (This is stricter security than TPM which considers the whole box to be the physical security perimeter.) Hence, SP provides automatic encryption and hashing of all temporary data belonging to the TSM if its cache lines get evicted from the microprocessor chip. Within the microprocessor chip, the TSM's secure cache lines are decrypted, so that there is no performance penalty with respect to the processor's pipelined execution. These secure lines are tagged (as shown in Figure 7). Further, SP hardware protects the general register values of the microprocessor, on interrupts, by automatic encryption and hashing, before allowing the OS to handle register saving. (This GR hash value and the interrupt return address is securely stored in new SP registers on-chip, as shown in Figure 7) Hence, the attacker can not get any access to plaintext information of the TSM during its execution, or while the TSM process is suspended or resumed. This protects leaking out bits of the DRK and SRH, as well as the secrets in the pervasive storage structure.

The TSM's code is integrity checked. SP provides dynamic Code Integrity Checking to TSM code on a cacheline granularity during runtime. Since this need only be done on the rare cache misses of the last level of on-chip caches, performance degradation is not noticeable. Prior to TSM installation on a given SP-enhanced machine, each instruction cache line chunk of TSM code is "signed" by generating a keyed-hash of that cacheline (minus the space to insert a 128-bit hash value) of instructions, keyed by the DRK. On a cachemiss, as the instruction cacheline for a TSM is brought into the microprocessor chip, its hash is re-computed. If it does not match the keyed-hash value stored at the end of the instruction cacheline, an exception is raised and that corrupted TSM instruction cacheline is not stored into an on-chip cache (which is typically the Level-2 or Level-3 cache). Hence, SP automatically provides protection of the TSM code from dynamic hostile code insertion, including those from buffer overflow attacks. It also provides detection of static hostile code modification while on the disk, by both the method just described and by verifying the integrity of the static image of the TSM code signed by the private key of the trusted software vendor (in this case the vendor of the trusted DBT).

We protect the entire DBT as a TSM in application space protected by SP hardware (Figure 6). We protect the code cache of the DBT as secure temporary data of the TSM that is automatically protected by encryption and hashing if any of it is evicted from the on-chip caches as the DBT translates or optimizes instructions. This means that attackers cannot see unencrypted contents of the code cache, and cannot modify the code cache without detection. Also, the integrity of the DBT code itself will be protected by SP's Code Integrity Checking mechanism. Hence, the application-level DBT itself, and its code cache, can be protected by SP from software and hardware attacks. In addition, the SP pervasive secure storage allows the DBT to keep secret information safe from attackers, across power off and on events.

Since the hardware additions for SP represent a very small area in a microprocessor chip, it is not difficult to add. (SP consists of only a few registers, 1-2 tag bits on a cacheline, and an encryption and hashing engine at the last on-chip cache miss handling mechanism.) These few hardware additions for security of the DBT can be done together with the hardware additions to speed up the dynamic translations done by a DBT.

6. Conclusions and Future Work

We have surveyed a range of software protection techniques and commercial software programs that can be used to enhance the security of programs, and described how they work. While many of the software techniques can fall into multiple categories, we have categorized them into 5 classes of techniques that: (1) prevent buffer overflow attacks, (2) prevent code modification attacks, (3) protect system calls, (4) mitigate reverse engineering and (5) check security assertions. Within these 5 classes, we described 12 representative software protection techniques.

We examined whether a DBT, in particular the StarDBT, inadvertently removes or lessens these software protections when the DBT applies its standard translation and optimizations to the application programs. Our results show that almost all of the software security protections are preserved, except for 4 cases where care has to be taken and certain changes in the DBT may be necessitated (see Table 1 and section 4).

Since DBTs are often application-level programs, they can themselves be attacked. Hence, we also proposed three solutions to protect the DBT itself. In particular, we show that the Secret Protection (SP) processor features can protect the application-level DBT and its code cache.

Future work can examine more software protection techniques, different types of DBT translation and optimizations, and DBT changes or features that can preserve the security of all essential software protection techniques. Some software protections may be superfluous – especially if the DBT is enhanced to provide these protections itself. Future work will also explore the many ways a trusted DBT can provide security enhancements for all programs – even legacy programs with no software security protection at all. In addition, future work can also perform a detailed security analysis of the mapping of the DBT and its code cache to a Trusted Software Module protected by the SP hardware.

References:

1. Adams, K. and Agesen, O. 2006. “A comparison of software and hardware techniques for x86 virtualization.” In *Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems* (San Jose, California, USA, October 21 - 25, 2006). ASPLOS-XII. ACM, New York, NY, 2-13.
2. Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I., "SWIFT: software implemented fault tolerance,"*, Proceedings of International Symposium on Code Generation and Optimization*, vol., no., pp. 243-254, 20-23 March 2005
3. Feng Qin; Cheng Wang; Zhenmin Li; Ho-seop Kim; Yuanyuan Zhou; Youfeng Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on* , vol., no., pp.135-148, Dec. 2006
4. Wu, Q., Martonosi, M., Clark, D. W., Reddi, V. J., Connors, D., Wu, Y., Lee, J., and Brooks, D. 2005. “A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance.” In *Proceedings of the 38th Annual IEEE/ACM international Symposium on Microarchitecture* (Barcelona, Spain, November 12 - 16, 2005). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 271-282.
5. Wang, C., Hu, S., Kim, H.-s., Nair, S., Breternitz, M., Ying, Z., and Wu, Y. “StarDBT: An Efficient Multi-platform Dynamic Binary Translation System.” *ACSAC’07: Asia-Pacific Computer Systems Architecture Conference*. pp. 4—15. 2007, Seoul, Korea.

6. Dwoskin, J. S. and Lee, R. B. 2007. “Hardware-rooted trust for secure key management and transient trust.” In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA, October 28 - 31, 2007). CCS '07. ACM, New York, NY, 389-400.
7. Lee, R. B., Kwan, P. C., McGregor, J. P., Dwoskin, J., and Wang, Z. 2005. “Architecture for Protecting Critical Secrets in Microprocessors.” In *Proceedings of the 32nd Annual international Symposium on Computer Architecture* (June 04 - 08, 2005). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, 2-13.
8. Vendicator, Stackshield, available at http://www.angelfire.com/sk/stackshield/
9. Propolice, “GCC extension for protecting applications from stack-smashing attacks”, available at http://www.trl.ibm.com/projects/security/ssp/
10. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” In *Proceedings of the 7th USENIX Security Symposium*, pages 63--78, San Antonio, TX, January 1998.
11. Libsafe, Avaya Labs Research, available at http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails
12. C. Kil, J. Jun, C. Bookholt, J. Xu and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, 2006, pp. 339-348.
13. Horne, B., Matheson, L. R., Sheehan, C., and Tarjan, R. E. 2002. “Dynamic Self-Checking Techniques for Improved Tamper Resistance.” In *Revised Papers From the ACM Ccs-8 Workshop on Security and Privacy in Digital Rights Management*
14. R. El-Khalil and A.D. Keromytis, "Hydan: Hiding information in program binaries," *Proc. 6th Int'l Conf. Information and Communications Security (ICICS'04)*, LNCS 3269, pp.187–199, Springer, 2004.
15. Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A., “A Sense of Self for Unix Processes”, In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (May 06 - 08, 1996)
16. [8LGM]. [8lgm]-advisory-16.unix.sendmail-6-dec-1994. http://www.8lgm.org/advisories.html.
17. “Injection flaws”, Open Web Application Security Project (OWASP), available at http://www.owasp.org/index.php/Injection_Flaws
18. Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, Richard D. Schlichting, "System Call Monitoring Using Authenticated System Calls*," IEEE Transactions on Dependable and Secure Computing* ,vol. 3, no. 3, pp. 216-229, July-September, 2006.
19. Christian S. Collberg and Clark Thomborson. “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.” In *University of Arizona Technical Report 2000-03*, Feb 2000.
20. Ogiso T, Sakabe Y, Soshi M, and Miyaji A. “Software Obfuscation on a Theoretical Basis and Its Implementation.” In *IEICE Trans Fundam Electron Commun Comput Sci (Inst Electron Inf Commun Eng),* Vol.E86-A;No.1;pp..176-186(2003).
21. Dotfuscator. PreEmptive Solutions. http://msdn.microsoft.com/en-us/library/ms227240%28VS.80%29.aspx
22. “Obfuscated code”, Wikipedia, available at http://en.wikipedia.org/wiki/Obfuscated_code.
23. Christian Collberg, Clark Thomborson, and Douglas Low. “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.” In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98)*, January 1998.
24. Christian Collberg, Clark Thomborson, and Douglas Low. “Breaking abstractions and unstructing data structures.” In *IEEE International Conference on Computer Languages (ICCL'98)* , May 1998.
25. EXECryptor, available at http://www.strongbit.com/execryptor_inside.asp
26. Lallous, “Detect if your program is running inside a Virtual Machine”, available at http://www.codeproject.com/KB/system/VmDetect.aspx
27. Kc, G. S., Keromytis, A. D., and Prevelakis, V. 2003. “Countering code-injection attacks with instruction-set randomization.” *In Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA, October 27 - 30, 2003). CCS '03. ACM, New York, NY, 272-280.
28. George Necula, “Proof-carrying code”, In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

29. Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skalesky, A., Wang, Y., and Zemach, Y. "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems." In *MICRO'36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. pp. 191—201. 2003, San Diego, CA, USA.
30. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser. A., Lowney. G., Wallace, S., Reddi, V., and Hazelwood, K. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. pp. 190—200. 2005, Chicago, IL, USA.
31. J.Wilander and M. Kamkar. "A comparison of publicly available tools for dynamic buffer overflow prevention." In NDSS, Feb 2003.
32. Lu, J., Chen, H., Fu, R., Hsu, W., Othmer, B., Yew, P., and Chen, D. 2003. "The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System." In *Proceedings of the 36th Annual IEEE/ACM international Symposium on Microarchitecture* (December 03 - 05, 2003). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 180.
33. Bruening, D. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation." Ph.D thesis, Massachusetts Institute of Technology, 2004.
34. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J. "The Transmeta Code Morphing$_{TM}$ Software: Using Speculation, recovery, and Adaptive Retranslation to Address Real-Life Challenges." *Proceedings of the International Symposium on Code Generation and Optimization*. pp. 15—24. 2003, San Francisco, CA, USA.