Towards Integral Binary Execution: Implementing Oblivious Hashing Using Overlapped Instruction Encodings

Matthias Jacob
Nokia Research Center
United Kingdom
majacob@nokia.com

Mariusz H. Jakubowski
Microsoft Research
Redmond, WA
mariuszj@microsoft.com

Ramarathnam Venkatesan
Microsoft Research
Redmond, WA (USA)
venkie@microsoft.com

ABSTRACT

Executing binaries without interference by an outside adversary has been an ongoing duel between protection methods and attacks. Recently, an efficient kernel-patch attack has been presented against commonly used self-checking code techniques that use checksumming ahead of execution. While methods based on self-modifying code can defend against this attack, such techniques depend on low-level architectural details and may not be practical in the long run. An alternative defense is to use oblivious hashing (OH). Instead of checking code integrity prior to execution, OH can verify untampered runtime behavior continuously. However, earlier OH approaches have some weaknesses, particularly with binary code: Physical instruction bytes cannot be easily checked during execution, and an attacker may be able to detect and remove OH checks, since OH alone does not provide tamper-resistance or obfuscation.

In our approach, we deliberately overlap a program's basic blocks so that they share instruction bytes. This increases tamper-resistance implicitly because malicious modifications affect multiple instructions simultaneously. Also, our scheme facilitates explicit anti-tampering checks via injection of OH instructions overlapped with target code, enabling OH that can verify integrity of both runtime state and executing instructions. Thus, our method addresses anti-checksum attacks without resorting to self-modifying code, and also extends OH to verify physical code, not only program state. In addition, overlapping facilitates resistance against disassembly and decompilation. Our approach works on processor architectures and byte-codes that support variable-length instructions. To our knowledge, this is the first technique that blends tamper-resistance into architecture and therefore significantly improves robustness of binaries.

Categories and Subject Descriptors

D.m [**Software**]: Miscellaneous—*Software protection*; C.5.0 [**Computer Systems Organization**]: Computer System Implementation—*General*; D.2.11 [**Software Engineering**]: Software Architectures—*Information hiding*

General Terms

Algorithms, reliability, security

Keywords

Oblivious hashing, tamper-resistance, integrity checking, obfuscation, overlapped code, anti-disassembly

1. INTRODUCTION

Protecting media players against deliberate attacks is becoming increasingly important on today's Internet. Software pirates often compromise player software to remove copy-protection and watermarks from digital content. In addition, computer viruses turn normal PCs into malicious attack hosts that hit the net almost every day, thereby slowing down servers and network access.

On open platforms such as PCs, virtual memory provides protection among processes, but an adversary with access to the OS kernel is able to take over and tamper with any process. To protect against these attacks in software, various methods for tamper-resistance have been developed (e.g., [8, 13, 26]). These turn a program P into an equivalent tamper-resistant program $P' = O(P)$, which detects manipulation attempts. However, P' is commonly based on techniques that verify code integrity before execution takes place. With full access to the PC, an attacker can circumvent these protection schemes – e.g., by tampering with the code after the integrity check takes place, or by executing tampered code while passing original code to checksum and hash routines [47]. Self-modifying code can protect against this type of attack [25], but may not always be viable, since low-level operations can be obstructed by specific architectural features (e.g., the execute-only bit on the x86 platform).

In contrast to other checksumming techniques, oblivious hashing (OH) does not verify machine-code bytes, but computes input-specific hashes based on program state during execution [29, 14, 35]. After initializing a hash, OH typically updates it with results of variable assignments, as well as with unique identifiers based on executed branches. At various points in a program, OH verifies the current hash against a pre-computed correct hash. Alternately, OH may compare hashes computed by two or more individualized replicas of the same code.

While OH has been implemented for high-level code [14] and Java byte-code [35], OH has seldom been investigated on low-level binary code. Such code has inherent properties that complicate reverse engineering, thus providing leverage for tamper-resistance when coupled with source-level methods. In particular, anti-disassembly techniques have drawn some attention recently [19]. Lack of correct disassembly increases the difficulty of reconstructing source code and tampering with execution. However, previous anti-disassembly approaches have suffered from some vulnerabilities [31].

In this paper, we pursue a novel direction and approach the problem of tamper-resistance from an architectural perspective that goes beyond anti-disassembly. As a central element of our methods, *code overlapping* allows pervasive sharing of code on different levels. This technique not only hinders code tampering, but also provides a significant improvement for anti-disassembly methods. In particular, we analyze and address the problem of quick disassembly resynchronization, which is a serious drawback of earlier techniques.

1.1 Overlapped Instruction Encodings

Code overlapping uses two different techniques:

- *Code interleaving* creates a new overlapped code block from two basic blocks by injecting additional redundant instructions. This overlapped block has two different entry points, one for each original block, and also hides one of the basic blocks from disassembly. To ensure security against substitution attacks [40], we protect the basic-block edges in the control-flow graph with computed branches that are hard to invert (e.g., via opaque predicates [18]).

- *Code outlining* increases the level of shared code in a program by creating a new function for two or more operationally equivalent code blocks. Any instruction the adversary changes in the outlined code modifies execution along two or more distinct control-flow paths. This creates problems for an attacker wishing to patch only one path.

We propose code overlapping as a novel comprehensive design for a binary-level tamper-resistance system. Our aim is to extend OH-based integrity checks to binaries and byte-codes in a secure and practical manner. Like standard OH, our construction can hash pure program state, but code overlapping adds the capability to include code bytes in the hash. This combines traditional code-byte checksums [13, 26] and program-state hashes, enabling more comprehensive tamper-resistance. In addition, our approach offers a method to hash code bytes without explicit access to physical code segments in memory. As with other integrity-checking methods, however, our proposal is intended as just one building block of an effective protection system.

Our main contributions are as follows:

- We find that commonly used anti-disassembly methods are not effective, even against a weak adversary, and require significant overhead. In particular, two instruction sequences usually resynchronize after a few instructions because of the Kruskal Count [32].

- We analyze limitations of previous approaches to OH and roadblocks against effective usage of code sharing in common binary code.

- We introduce two techniques for code overlapping – code outlining and code interleaving – that enable sharing of code bytes on x86 and byte-code platforms to protect against disassembly and tampering.

- In addition to checking machine state, we use code overlapping to check code in memory during execution. This is an important step to prevent anti-checksum attacks without resorting to self-modifying code [47].

- Via a tool implementation, we show how to use code overlapping to realize secure OH for x86 native code. Using the SpecCPU2000 benchmark suite, we verify the practicality of this approach.

2. OBLIVIOUS HASHING FRAMEWORK

Oblivious Hashing (OH) is a state-of-the-art technique to protect program execution against tampering. OH transforms programs by injecting code that computes hashes over deterministic runtime state. After hash initialization at chosen program locations, OH updates hashes with variable assignments and unique identifiers based on runtime control flow. At various points during execution, a verifier checks hashes against pre-computed values to detect tampering an adversary may have undertaken [29, 14, 35]. Alternately, the verifier may compare hashes computed by two or more individualized code replicas.

OH alone does not provide a cryptographic security model with formally analyzable properties, but complicates manual and automatic reverse engineering heuristically. We assume that an adversary can use disassemblers, debuggers, and other tools for both static and dynamic analysis. This is similar to most practical software-protection approaches that undergo the typical design-and-attack cycle (e.g., [28, 15]). Even in cases where some security modeling and analysis are given [44], it is often possible in practice to avoid the need to solve any stipulated hard problems. Nonetheless, OH can serve as a crucial building block in certain tamper-resistance frameworks designed for analyzable security [20].

OH and other techniques are useful as heuristic building blocks of comprehensive protection systems, with real-life security determined by quality of engineering, implementation, and penetration testing. More formally, given instructions $I = \{i_0, i_1, ..., i_n\}$ of a program, we compute an oblivious hash over the instruction counter C, memory state M, and input parameter P. For the execution trace T, let $H(T)$ denote the hash:

$$H(T) = H(I,\, M,\, C^0,\, M^0,\, P)$$

where C^0 and M^0 are the initial instruction counter and memory state, respectively. If $H(T)$ is not a valid hash, additional injected code in the program will detect this after hash computation.

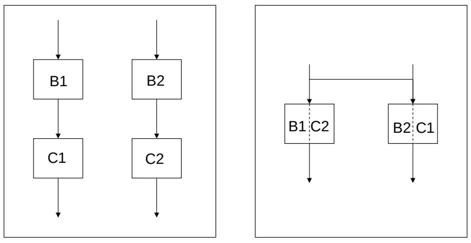
Figure 1 shows an example of OH at the source-code level. The main idea is to initialize a hash value and subsequently update it upon every assignment and control-flow transfer in the code. For an assignment, we update the hash by combining it with the value being assigned; for a control-flow transfer, we combine the hash with a unique basic-block identifier. The update operation may be accomplished via some combination of bitwise and arithmetic operations, or via a cryptographic hash function, such as SHA-256. While the former has better performance and is easier to blend in, the latter may be preferable for security reasons, though attacks in practice likely will not depend on cryptanalyzing the hash. Finally, a hash may be verified against a pre-computed "correct" hash at various chosen locations in the code. Alternately, to avoid the need for hash pre-computation, two or more (individualized) replicas of

```
int x = 123;

if (GetUserInput() > 10)
{
    x = x + 1;
}
else
{
    printf("Hello\n");
}
```

```
INITIALIZE_HASH(hash1);

int x = 123;
UPDATE_HASH(hash1, x);

if (GetUserInput() > 10)
{
    UPDATE_HASH(hash1, BRANCH_ID_1);
    x = x + 1;
    UPDATE_HASH(hash1, x);
}
else
{
    UPDATE_HASH(hash1, BRANCH_ID_2);
    printf("Hello\n");
}

VERIFY_HASH(hash1);
```

1: Oblivious hashing in source code: Oblivious hashing injects hashing instructions into the original program (top). The resulting code is shown on the bottom.

the same code can compute and compare hashes, flagging a tampering attempt if hashes fail to match.



2: Checking using overlapped code: OH without overlapped code requires checks to be in sequence with basic blocks (left). In OH that uses overlapped code, the checker can be interleaved with another basic block, thus protecting it.

The idea in code overlapping is to tie checking of one basic block to execution of another basic block. Figure 2 shows the scenario. On the left side, without code overlapping, blocks B1 and B2 have checkers C1 and C2, respectively. However, with code overlapping as shown on the right, the checkers are strongly interleaved with the other basic blocks. Instructions of B1 and state of B2 are checked during execution of B2; also, instructions of B2 and state of B1 are verified during execution of B1. Compared to common approaches for checking, code overlapping has the following advantages:

- *In-place static code-byte checks:* The checker does not need to load instruction bytes explicitly from memory, which may not even be accessible in the case of byte-code interpreters. In code overlapping, checks have the form '`mov eax, 0x81A1903821; call _check`' instead of '`mov esi, _start; mov eax, ds:esi; call _check`', where `_start` denotes the start of the code segment and `_check` the checker function. The immediate operand of the first checking sequence contains the code bytes at `_start`. This prevents attacks in which an adversary monitors memory reads of the code bytes, since the code bytes are essentially "read" during execution.

- *Interleaved basic blocks:* During execution of one basic block, at least one other block is checked. An interleaved block has multiple entry points (one for every original block), and execution automatically invokes the checkers. Separating the original basic blocks from the checkers is difficult, because this also repositions code bytes of the original basic block.

- *Computed branches hiding edges of the control-flow graph:* Given an interleaving of basic blocks B1 and B2, execution obtains the correct address of B1's runtime follower only when the code of B2 has not been tampered and the program state during execution of B1 is correct. This significantly limits an adversary's attacks, because code cannot be easily moved. When the instruction pointer EIP is incorrect for the computed jump to the follower, the program will likely malfunction or crash.

3. CODE OVERLAPPING

Code overlapping can share entire instructions or code blocks, which may be outlined into separate functions that are called from two or more distinct control-flow paths; we refer to this as *semantic overlapping*. At a lower level, *physical overlapping* shares bytes between two (or more) instruction encodings (e.g., via code interleaving). Physical overlapping requires variable-length instructions and control-flow transfers to byte-aligned addresses, as supported by today's popular architectures. However, semantic overlapping is architecture-independent and may be done at both the binary and source-code levels.

3.1 A Simple Method for Code Overlapping

The simplest method for sharing instruction bytes by overlapping is to inject fake instructions into the code. During tamper-free execution, a fake instruction is never executed. However, when an adversary tampers with the code, execution may encounter these fake instructions and cause the application to malfunction.

Figure 3 shows how code overlapping with fake instructions works: The process takes the original code snippet on the top and injects two bytes `B8 01` to generate a fake instruction. In addition, an inserted branch instruction preserves the original execution order. During a correct execution without tampering, the fake `mov` instruction is always unreachable, because the outcome of the `jne` instruction always circumvents it. Since the branch target is not the address of any instruction in the straight-line disassembly, the adversary may assume that the branch is not taken and may not find the protected *shl* instruction.

This approach is also a popular technique against correct disassembly [19]. The straight-line disassembly of the previous example code never shows the protected *shl* instruction in the clear.

```
I:   C1 E1 02  shl eax, 2
     41        inc eax
O:   C3        ret
```

```
I:    3D FF 00 00 00   cmp eax, 0xff
      EB 02            jne X
I1:   B8 01            mov eax, ...
X:    C1 E1 02         shl eax, 2
      90               nop
      41               inc eax
O:    C3               ret
```

```
I:    3D FF 00 00 00      cmp eax, 0xff
      EB 02               jne <invalid>
I1:   B8 01 C1 E1 02 90   mov eax, C1E10290
      41                  inc eax
O:    C3                  ret
```

3: Code overlapping with fake instructions: Code overlapping injects fake instruction into the original instruction sequence (top), resulting in the code sequence in the middle. An adversary sees the broken instruction sequence on the bottom.

However, it is relatively easy to circumvent this protection – a disassembler simply needs to probe different byte offsets following a suspicious branch instruction [31].

In addition, the number of instructions that code interleaving is able to protect with fake instructions is very limited. Disassembly resynchronizes very quickly – in only 2-3 instructions on average – because of a mathematical phenomenon called the Kruskal Count [32]. This models resynchronization in a random sequence of instruction lengths. Initially, disassembly and program execution may be off by k bytes. However, when a disassembler continues decoding instructions in a straight-line sequence, the instruction streams resynchronize after an expected $O(\log k)$ steps.

3.2 Resynchronization of Disassembly

At first glance, constructing overlapped code seems to be an art by itself. First of all, generating instructions that are part of other instructions is limited on most architectures. In general, this is only possible with variable-length instruction sets like x86. Even after constructing one overlapped instruction, the next one needs to follow soon in the code, because the code will eventually resynchronize. This has been assumed in previous work on anti-disassembly [19, 31], but here we give a precise mathematical explanation – The Kruskal Count.

The Kruskal Count is best explained as a card trick: Given a shuffled card deck, a magician asks a subject to choose a secret number between 1 and 10. The magician turns around card after card, and every time the subject decreases the counter by 1. When the count reaches 0, the first tapped card occurs in the stack, and the subject sets the counter to the value on the card. The magician also secretly picks a number and follows the same procedure as the subject. They continue with this procedure until the card deck is exhausted. Surprisingly, they eventually both end at the same card, and the magician is able to "guess" the subject's last tapped card. In code interleaving with fake instructions, the game is similar. The correct offset for disassembly is hidden in the code, while disassembly begins at a different nearby offset. When the instruction lengths are drawn unpredictably, as is the case with compiler-generated code, the correct and actual disassemblies eventually resynchronize. The question is how we can achieve the best strategy to prevent this resynchronization.

The model of the Kruskal Count is similar to a leapfrog game on two differently shuffled card decks. Player 1 has a white leapfrog and player 2 has a black leapfrog. Both leapfrogs mark the next tapped card. We now use a Markov Chain to model the game – the difference between the white and the black leapfrog is basically the state information of the Markov chain. If we model the card deck as a uniformly distributed card deck as shown in [32], the probability for resynchronization at step t after N cards of a card deck that has B different values is about

$$P[t > N] = e^{-\frac{4}{B^2}} + O\left(\frac{1}{B^3}\right)$$

when $N \to \infty$. This is basically the distribution for the coupling time of a rapidly mixing Markov chain, and it takes about $f_R(B) = \frac{B^2}{16}$ steps until resynchronization on average. The is the *resynchronization frequency* f_R that depends solely on B, the average number of bytes per instruction. The disassembly code resynchronizes every f_R^{-1} instructions. For further details, we advise the interested reader to investigate the very involving paper on the Kruskal Count by Lagarias et al [32].

3.3 Code Interleaving

The theoretical limitation in the Kruskal Count complicates the task of efficiently implementing strong anti-disassembly techniques via overlapped code, because such code requires many indirect jumps. In addition, this anti-disassembly is weak and can be easily broken [31]. Therefore, we extend idea of overlapping to enhance both anti-disassembly and tamper-resistance by code interleaving.

Simply sharing code generally improves tamper-resistance, because an adversary modifies multiple control-flow paths when changing a single instruction. Code interleaving pushes the idea of code sharing further, and shares instruction sequences at a finer granularity. Similar to memory-space-preserving implementations of error tables in Microsoft Basic on the Altair, interleaved instructions even mutually share their instruction bytes [24].

In contrast to our earlier anti-disassembly example, we take a new approach to interleaving instructions and explicitly construct overlapped code using misaligned instructions. Instead of injecting only fake instructions, code interleaving includes these instructions in the actual execution, thereby making it hard to distinguish the fake instruction from a real one. The branch condition depends on an opaque predicate to protect against static analysis.

Instead of overlapping the instruction stream with fake instructions, code interleaving mutually overlaps two basic blocks `B1` and `B2` of the program into a *super-block* with multiple entry and exit points. This construction never resynchronizes; moreover, when an adversary tampers with the super-block, at least one other basic block of the program breaks. Furthermore, basic blocks are hidden within the super-block, and an adversary needs to locate all the different entry and exit points to discover all hidden basic blocks.

The essential idea of this integrity-checking method is to interleave (or merge) two instruction sequences in a program, creating a code block that contains both sequences and two corresponding entry points. The scheme injects jump instructions to preserve original sequential execution when the program reaches one of the entry points. The method subsequently replaces these jump instructions with special hashing instructions that maintain the same control flow.

We illustrate this process with the following example in an abstract assembly pseudocode. We assume an assembly language of instructions encoded via variable-length byte sequences, along with an architecture that supports control transfer to arbitrary byte addresses (i.e., we can jump into the middle of instructions). Consider the two instruction sequences below:

```
SEQ1:  INST_1
       INST_2
```

```
SEQ2:  INST_A
       INST_B
```

SEQ1 and SEQ2 denote address labels, while (INSTR_1, INSTR_2) and (INSTR_A, INSTR_B) represent arbitrary instructions in some assembly language or custom byte-code. For clarity, each sequence contains only two instructions. The interleaving procedure creates the code block below:

```
SEQ1:  INST_1
       JMP L2
SEQ2:  INST_A
       JMP LB
L2:    INST_2
       JMP L3
LB:    INST_B
L3:
```

Depending on the entry point into this block (SEQ1 or SEQ2), one of the original instruction sequences will execute. The next step is to replace the jump instructions with special hash instructions that do not affect control flow. For example, JMP L2 may be replaced by an instruction whose size in bytes matches the sum of the lengths of JMP L2, INST_A, and JMP LB. In the resulting execution path below, HASH_1 denotes such an instruction:

```
SEQ1:  INST_1
       HASH_1
       INST_2
       HASH_2
```

```
SEQ2:  INST_A
       HASH_A
       INST_B
```

The exact byte encodings of instructions now become important. We assume that HASH_1 actually contains the bytes of INST_A, and the address specified by SEQ2 lies somewhere within the encoding of HASH_1. Consequently, transferring control to SEQ2 should yield the execution path in SEQ2 on the right-hand side.

As part of their encodings, the hash instructions HASH_1 and HASH_A contain bytes of the original instructions INST_A and INST_2, respectively. The hash instructions also may or may not operate on program state, such as working registers and memory locations. Thus, execution of the hash instructions can effectively update hash values based on either the code bytes or both code bytes and state. Note that this yields code-byte checksums without explicit reads of code bytes.

As with OH, integrity checks may be performed by verifying runtime hash values. However, our approach also makes patching more difficult by using the same bytes to encode two (or more) instructions. Thus, attempting to tamper with a particular instruction may inadvertently alter other instructions, creating problems for a hacker.

To achieve the overlap properties described above, suitable instruction encodings are necessary. Under the x86 architecture, we have found that usable encodings exist to implement a variety of possible hash instructions, as described later. However, to maximize the effectiveness of overlapping instructions, we may also design our own custom byte-code geared towards self-checking and other desired security properties [7].

```
B1:
I10:  C1 E0 02   shl eax, 2
I11:  40         inc eax
      C3         ret

B2:
I20:  48         dec eax
I21:  C1 E8 03   shr eax, 3
      C3         ret
```

```
I10:  C1 E0 02   shl eax, 2
      EB 03      jmp I11

I20:  48         dec eax
      EB 04      jmp I21

I11:  90         nop
      40         inc eax
      EB 03      jmp O

I21:  C1 E8 03   shr eax, 3
O:    90         nop
      C3         ret
```

4: Interspersing step in code interleaving: Code interleaving takes the original basic blocks on top and alternates their instructions in the merged basic block on the bottom.

The above construction consists of two basic steps – *interspersing* and *merging*. Figure 4 shows the interspersing step in constructing interleaved code. This step combines consecutive instructions from B1 and B2, alternating between the blocks, and injects branch instructions to maintain original instruction sequencing. When execution arrives at I10:, the program executes the shl instruction and branches to I11:. Alternately, when execution arrives at I20:, the program executes the dec instruction and branches to I21:.

Op.-len.	Mask-instr.	Mask-op.	Opcode size	Inst.-size
1 byte	and al,...	24 ...	1 bytes	2 bytes
2 bytes	and ax,...	66 25 ...	2 bytes	4 bytes
4 bytes	and eax,...	25 ...	2 bytes	6 bytes
8 bytes	and [...],...	23 ...	1 bytes	9 bytes

5: Masking instructions: The number of opcode bytes of the original instruction followed by the best mask instruction, and the total number of opcode bytes for the whole masked instruction. The mask instruction is only an example; many different but equivalent mask instructions are possible.

In the merging step, code interleaving replaces the interspersed branch instructions with overlapping instructions that mask out the instructions from the other basic block. Figure 5 shows examples for the different instruction patterns that can be used to mask overlapped instructions. For the last class of masking instructions, the code needs to handle exceptions to proceed to the next valid instruction. On AMD-64, it is even possible to use up to 64 bits in the immediate operand. In the last instruction category, this leaves up to 12 bytes to be used for overlapped code.

In addition, code interleaving needs to align instructions with NOPs for padding to the next available operand length. During the instruction selection process for the overlapping instructions, a register allocator may need to spill registers into memory, but this paper does not address optimizing performance by limiting the amount of registers used in these masked instructions.

```
I10:  C1 E0 02   shl eax, 2
      81 F1      xor ecx, ...
I20:  48         dec eax
      81 E9      sub ecx, ...
      90         nop
I11:  40         inc eax
      81 C1      add ecx, ...
I21:  C1 E8 03   shr eax, 3
      90         nop
O:    C3         ret
```

```
B1:
I10:  C1 E0 02            shl eax, 2
      81 F1 48 81 E9 90   xor ecx, 90e98148
I11:  40                  inc eax
      81 C1 C1 E8 03 90   add ecx, 9003e8c1
O:    C3                  ret
```

```
B2:
I20:  48                  dec eax
      81 E9 90 40 81 C1   sub ecx, c1814090
I21:  C1 E8 03            shr eax, 3
      90                  nop
O:    C3                  ret
```

6: Merging step in code interleaving: In the final step code interleaving replaces the branches with overlapping instructions to merge the basic blocks into one super-block (middle). The original basic blocks have different entry points `I10` and `I20` into the super-block, respectively (bottom).

Figure 6 shows how code interleaving merges two basic blocks into one super-block. The super-block on the left-hand side contains all instructions, including the overlapping instructions. The actual execution on the right-hand side traverses only instructions of either `B1` (top) or `B2` (bottom).

It is straightforward to apply code interleaving iteratively. Instead of a basic block, code interleaving can also process a super-block and a basic block to generate a new super-block. However, this technique is not a silver bullet. The maximum number of basic blocks that we are able to generate iteratively depends on the maximum number of bytes per instruction. For example, if the maximum instruction length is six bytes, it is possible to overlap at most six basic blocks within one super-block.

3.4 Code Outlining

The goal of code overlapping is to share code among multiple parts of a program. Code interleaving showed us how to merge two basic blocks to make them dependent on each other. However, this is most effective when these blocks are frequently used on active program paths. To help with this, we use semantic overlapping, which we show how to implement via code code outlining.

As a generalization of tail merging, code outlining generates shared code on the function level. General outlining is not limited to procedure epilogues. Similar to common-subexpression elimination, code outlining locates common instruction sequences throughout the program, moving them into a new replacement procedure shared by all callers of the instruction sequence. This operation is essentially the inverse of inlining. An instruction is *outlineable* if it is part of a common instruction sequence, and at least one other duplicate exists somewhere in the program. As with inlining, code outlining can take place locally within the same procedure or globally across multiple procedures.

```
I:  83 C0 01   add eax, 1
    D1 E0      shl eax
    33 C1      xor eax, ecx
    83 C0 01   add eax, 1
    D1 E0      shl eax
O:  C3         ret
```

```
F:  83 C0 01         add eax, 1
    D1 E0            shl eax
    C3               ret
I:  E8 F5 FF FF FF   call F
    33 C1            xor eax, ecx
    E8 EE FF FF FF   call F
O:  C3               ret
```

7: Code outlining: Original instruction sequence with repeating code patterns (top) versus outlined instruction sequence (bottom).

Figure 7 shows an original instruction stream that contains repeating code patterns. Code outlining detects common code patterns and moves them into a separate procedure. The process replaces the original code patterns by calls to this procedure. After code outlining, the original basic block looks similar to the one on the right-hand side in figure 7. Code outlining replaces the code patterns with calls to the replacement procedure `F`.

In most cases, code outlining works by simply moving the code from the program into the replacement procedure `F`. However, there are several cases that require special treatment. First, because of the call statement, the stack pointer is off by one position. Any code that accesses the stack pointer in `F` must be adjusted and rewritten. Similarly, when `F` contains a branch outside of `F`, code outlining needs to insert an instruction that adjusts the stack before the branch can take place. Second, the destination addresses of branches into instructions of `F` must be adjusted properly and turned into a call statement to ensure that execution returns to the right place. When the branches are indirect, code outlining needs some additional information from the compiler to find the correct instructions for rewriting.

Obviously, code outlining increases the connectivity in the CFG, because it increases the number of commonly used blocks. Unfortunately, code outlining operations are easily reversible – a common inlining tool can simply copy the replacement procedures back into the original instruction sequence. The code contains the mapping between the caller and the callee in the clear.

To hide these mappings, code outlining uses opaque predicates or computed branches [18]. The branch function $\Psi(x)$ is hard to invert, given the outcome of $\Psi(x)$. Given $\Psi(x_0) = 1$, it is difficult to find an x_0 that fulfills the condition; in the case of a computed branch, given an address y, it is hard to find an x_0 such that $\Psi(x_0) = y$. For example, given a one-way function $h(x)$ and

$\Psi(x_0) = 1$ iff $\Psi(x_0) = \{h(x) = x_0\}$, it is hard to invert $\Psi(x_0)$, because inversion of Ψ requires inversion of $h(x)$. Security of opaque predicates depends on the computational complexity that is necessary to invert them.

```
_jmp:   ...
        C3                 ret
I:      E8 F9 FF FF FF     call _jmp
J1:     E8 ...             call G
                           ...
J2:     E8 ...             call F
        C3                 ret
```

8: Code outlining with computed branches: The computed branch `_jmp` stops the static analysis tool and prevents the outlined code in `F` to be inlined again.

Figure 8 shows the construction of code outlining with a computed branch function `_jmp`. First, program execution encounters the call to `_jmp` and computes the branch destination, which in this case is either `J1` or `J2`.

Computed branches protect code outlining against static-analysis tools. The only way to determine call destinations efficiently is exhaustive testing of the outlined program P'. This depends only on the dynamic input-output behavior of the original program P and is beyond the scope of this paper.

Generally, code outlining has its weaknesses as a standalone technique, because it relies heavily on individual code characteristics. However, outlining can significantly complicate the CFG of the program, thus increasing the security of code overlapping. In combination with peephole optimizations, outlining becomes more powerful, but this is beyond the scope of this paper.

3.5 Applications of Code Overlapping

3.5.1 *Overlapping Instruction Sequences in Oblivious Hashing*

This section provides some examples of using overlapping instruction sequences in OH. Overlapping instruction sequences have two goals, namely protecting overlapped code against detection by static analysis, as well as protecting overlapped code (and therefore the hashing code) against tampering.

Checksumming Code.

The first example for an overlapping instruction sequence is code that computes checksums over a basic block. Instead of jumping around the instruction of the second basic block of the interleaved block, the code uses a spare register to compute a checksum over the second basic block B_0 that is currently not executed. Using overlapping instructions on one of the two interleaved basic blocks B_0 or B_1, the code computes a checksum over the existing instructions of B_0, and verifies the checksum at the end.

Figure 10 shows an example of checksumming code. `C1` and `C2` contain code of B_0, and execution of B_1 starts at `START`. Summing up instruction bytes from B_0, the code compares the sum with the target value in `C4`. When the compare operation fails, the program runs into an infinite loop. A static-analysis tool does not discover the overlapping instructions as irrelevant instructions because of the def-use dependency. The disadvantage of this construction, however, is that an adversary can still override the `cmp` or `jmp` instruction to disable checksumming, therefore revealing a single point of failure for tampering.

```
ST:     BA                  mov edx,...
        ; load edx with bytes from C1
        ; and resume at C2

C1:     50                  push eax
        83 EC 01            sub esp, 1
        <code from B1>
C2:     81 C2               add edx,...
        ; add bytes from C3 to the checksum

C3:     83 C0 01            add eax, 1
        58                  pop eax
        <code from B1>
C4:     81 FA D3 43 EE 59   cmp edx,59EE43D3
        ; verify checksum in edx

END:    75 ...              jne ST
        ; if checksum is wrong, go to ST
```

9: Checksumming code: The overlapping instruction sequence contains the instructions that are necessary for computing the checksum of the basic block B_0 while B_1 is executed.

Self-Checking Code.

To eliminate the single point of failure, the overlapping instructions can verify matching instruction sequences and stop execution in case of a failure. This self-checking property gives real tamper-resistance that is bootstrapped within overlapped code, thus solving a problem that is usually hard to address via standard cryptography [8]. In self-checking code using overlapping instruction sequences, the general concept is simply to take one basic block B_0 and construct an interleaved super-block by using code patterns from B_0 repetitively.

Figure 10 shows an example on how to use overlapping instruction sequences for self-checking. In this example, we protect the `push eax; sub esp, 1` instruction sequence at `C3` from tampering. When execution starts at `START`, the program first loads the to-be-checked instruction bytes into `edx`. In `C2`, the program then executes the instruction sequence and compares the previously loaded instruction bytes at `C1` to the executed instruction sequence at `C3`. At `C5`, the code jumps back and executes `C3`. This construction requires an adversary to modify the code at `C1` and `C3` at the same time. However, compared to the earlier checksumming technique, no second basic block B_1 can be interleaved.

4. TOOL FOR CODE OVERLAPPING

To study the actual impact of code overlapping on binaries, we implemented a code-overlapping tool using Vulcan, a binary rewriting tool for Windows [42]. Vulcan is able to process any binary that includes a program's symbol tables. The output after binary rewriting is a normal binary that can be executed without any additional tools. The Vulcan-based tool reads in the source binary, processes the code, and writes the overlapped binary into a new file.

Code outlining and interleaving perform different operations, which can work together for better results. Code outlining slices a program P into smaller basic blocks that are more strongly interconnected than in the original program. On the other hand, code interleaving ensures transposition of outlined code into larger super-blocks that can be split up again.

Program evolution provides a useful iterative mechanism to achieve security. After alternating both techniques for a couple

```
ST:      BA              mov edx,...
         ; load edx with bytes from C1
         ; and resume at C2

C1:      50              push eax
         83 EC 01        sub esp, 1
C2:      81 C2           add edx,...
         ; add bytes from C3 to edx
         ; and resume at C4

C3:      50              push eax
         83 EC 01        sub esp, 1
C4:      74 FA           je C3
         ; je C3 should first be taken
         ; (to execute target code)
         ; and later fall through

         E8 02 00 00 00  call _jmp
         ; call may jump somewhere, based
         ; on results of comparison

END:     EB ED           jmp C1
         ; jmp C1 may never be reached;
         ; serves to hinder analysis
_jmp :   ...
```

10: Self-checking code: By injecting repetitive code patterns within the overlapped code, overlapping instruction sequences can realize self-checking code for tamper-resistance. The checked instructions are treated as both operands and opcodes during the same execution of the basic block.

of rounds, the resulting program P' becomes robust against low-cost tampering attacks. A random or secret input key q randomizes the instruction selection for code overlapping and basic-block selection in code interleaving.

Before the tool applies any of our code-overlapping techniques, it carries out a simple peephole-optimization step to minimize the number of bytes per instruction in the code. Note that on the x86 platform, the maximum size for an instruction in the operand of another instruction is 8 bytes. Due to this limitation, the number of blocks that can be overlapped simultaneously depends highly on the number of bytes per x86 instruction.

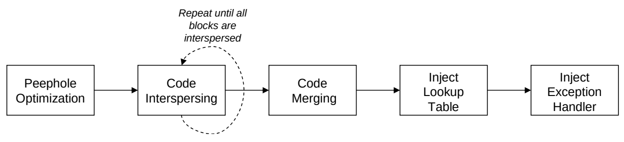
In the next step, the tool applies code outlining. This process accepts as input a program P, a secret q, and a security parameter p. Outlining first computes the maximum instruction length L, and then searches for all matching instruction patterns in P. Upon finding a match, the tool decides at random whether or not to outline the code pattern. If outlining takes place, the algorithm first checks whether the code pattern has already been outlined. If not, the tool copies the pattern into a replacement procedure and saves its address. Finally, outlining overwrites the instruction pattern with a call statement to the outlined replacement procedure.

The second main step, code interleaving, consists of two parts: *intersperse* and *merge*. The intersperse operation takes two basic blocks B_0 and B_1 and combines their instructions with interleaved branch instructions J_0 and J_1. These branch instructions separate the two instruction streams from B_0 and B_1, maintaining instruction sequencing in both blocks.

Our tool iterates over all basic blocks that have not yet been interspersed, picking two random blocks B_0 and B_1 and interspersing them into a single instruction stream B with interleaved jump instructions. After the tool has interspersed all basic blocks, it starts again with a random interspersed block B_0 and tries to determine whether the operands contain enough space to fit a second instruction. The construction is basically the same as for two blocks, but the jumps must instead traverse two basic-block instructions and two jumps, which could exceed the 8-byte field. With the AMD64 platform's larger operands, this happens quite frequently.

The merge operation transforms the instruction stream B into a single basic block. The algorithm first determines two registers R_0 and R_1 that are not live in the two previous basic blocks B_0 and B_1, respectively. The process then generates a sequence of overlapping instructions that use R_0 and R_1, and have length equal or less than J_0 and J_1. The main requirement for the sequence of overlapping instructions is that R_0 and R_1 be on a def-use chain, mainly to avoid detection by a static-analysis tool. A straightforward approach to accomplish this is to load and compare R_0 with the actual instructions and inject conditional branches into the code. The following section on applications of code overlapping contains more examples. After every overlapping instruction, the algorithm needs to pad instructions with `NOP` or other fake instructions that do not touch any live registers to fit them into the operand.

In our complete code-overlapping algorithm, the number of rounds N is a security parameter in addition to p. The overlapped program P' gains security when N and p increase, but also becomes slower. The `outline` and `interleave` steps do not commute. Once code has been interleaved, it cannot be outlined, and vice versa. Different security parameters q provide different tamper-hardened programs P', thus protecting against automated attacks and providing individualized tamper-resistance.



11: Code overlapping tool: The different steps for code overlapping in addition to code interleaving and code outlining.

Figure 11 shows the final layout of the tool. After the merging step, the tool injects the lookup table for the computed branches. To determine the jump target, the overlapped program first computes a cryptographic hash of selected registers, and then looks up the hash in the lookup table to find the destination address. Similarly, the exception handler looks up the instruction length at the current instruction and jumps directly to the next instruction. Extensive use of the exception handler should be avoided for performance reasons, so it may be advisable to restrict interspersing to instructions that do not exceed 4 bytes (minus 1 byte for the masking instruction byte of the next instruction) on x86.

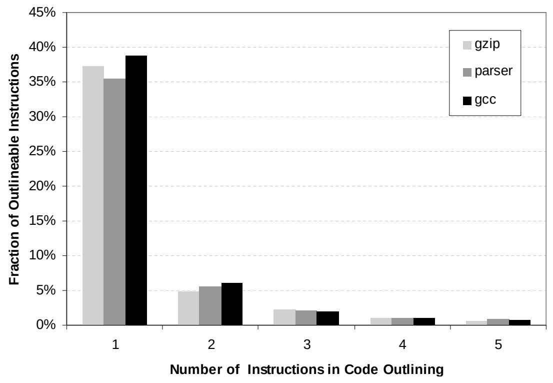
5. EVALUATION

Theoretically, security of code overlapping depends on subgraph connectivity in $CFG(P')$, the control-flow graph the adversary is able to determine from P'. The edge weight in the corresponding block-dependency graph $BDG(P)$ – which describes the number of edges for every basic block – increases linearly according to the number of edges in $CFG(P')$. Therefore, the main goal in code overlapping is to maximize the edge weights in $BDG(P')$. High connectivity in $CFG(P')$ makes it hard to tamper with P', because a single program manipulation causes multiple changes in different parts of the program. Since it is also hard for an adversary to reconstruct $CFG(P)$ from $CFG(P')$ using static-analysis tools, she is not able to reconstruct the original program P. In contrast, in programs with normal run-time checks, the additional checks increase only the number of basic blocks in the control-flow graph, and can be easily identified using static-analysis tools. We can always guarantee at least one interleaving step, increasing the connectivity of a vertex in $CFG(P)$ to at least 2; however, the strength of code overlapping depends on code heuristics.

For our analysis, we picked the SpecCINT 2000 suite [4]. SpecCINT is the industry standard for code performance and is representative for common code patterns that need to be protected in real code. To simplify the analysis, we included only three benchmarks: the compression engine `gzip` as a small benchmark with 6538 x86 instructions, `parser` as our medium-size benchmark with 20716 x86 instructions, and `gcc`, the C compiler, as a large benchmark. It has 275627 x86 instructions.

Code Outlining.

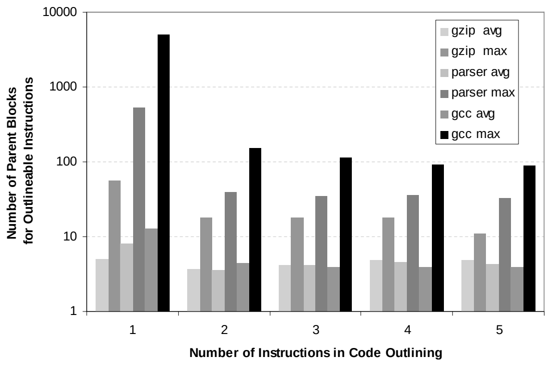
We first investigate the efficiency of code outlining. Note that since code outlining matches instruction sequences, it relies on code emitted by the compiler.

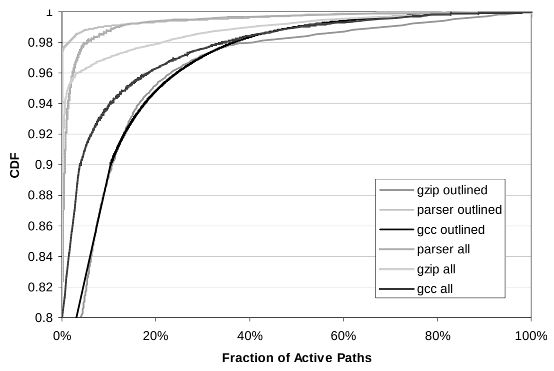


12: Outlineable instruction sequences: For single instruction simple code outlining without additional transformations already achieves 30% outlineable instructions.

Most importantly, we need to measure how many repeating instruction sequences the source binary contains. Figure 12 shows the total fraction of outlineable instructions for different instruction-sequence lengths. Without any additional code transformations, about 30% of all single instructions can already be outlined, but outlining happens to be very limited for any sequence of more than one instruction. To be truly resistant against tampering attacks, these outlined instruction blocks must be strongly connected. Figure 13 shows on the left graph the number of parent blocks for the outlined instruction blocks. Even though the average is usually around 5-10 parents, some blocks can have up to 5000 parents.

Tamper-protection of the overlapped binary also assumes that the outlined instruction blocks are on frequently used program paths; otherwise tampering cannot be detected immediately, and may even not be detected at all. The right graph in figure 13 shows the cumulative distribution function of the fraction of active paths in the program. In `gcc`, for example, 20% of all active paths are in about 94% of the program. Via Vulcan, we profiled the source binary and the overlapped binary, using the reference inputs from the Spec benchmarks. We also measured the number of basic blocks on active paths in the source binary, along with the number of outlined instruction blocks on such paths. As a result, we see that the outlined blocks are not in inactive areas of the program.





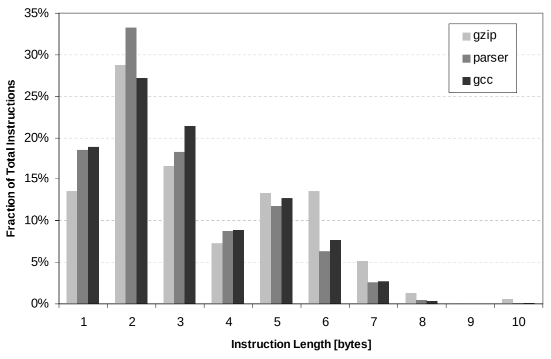
13: Outlined instruction sequences: These graphs show the number of parent blocks per outlined blocks (top) and the fraction of program paths that go through the blocks of the source binary and the outlined blocks (bottom).

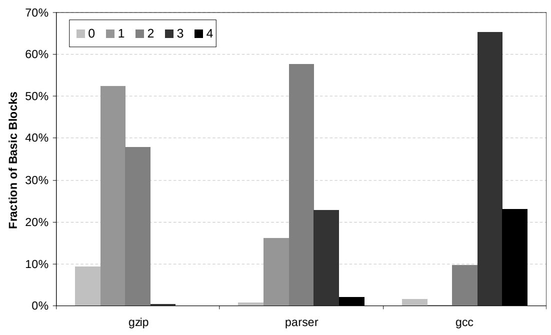
Code Interleaving.

In contrast to code outlining, code interleaving can process as many arbitrary instructions as possible, but the restriction is the maximum instruction length, since the address operand of the masked instruction is only 8 bytes on x86.

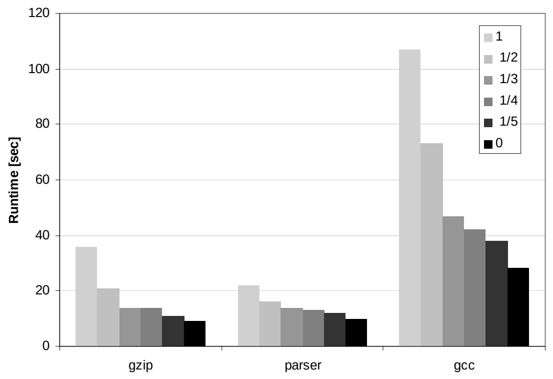
In our analysis of the actual code heuristics, we found that the average instruction length of the instructions in the three benchmarks ranges between 2 and 4 bytes. The left graph in figure 14 shows this result. Therefore, in our code overlapping tool, it is often possible to reuse a block at least once or twice in code interleaving – on AMD-64, even more often. The right graph in figure 14 shows the fraction of basic blocks that can be use for interleaving between 0 and 4 times during typical code-interleaving operations. Given that instruction operands can be up to 12 bytes on x86/AMD-64, and even without any additional optimizations, code interleaving can double or triple the connectivity and therefore the resistance factor in the overlapped binary.

However, the distribution alone does not answer the question on how fast the disassembly resynchronizes with the execution. For example, an instruction of length 1 does not necessarily resynchronize, but when multiple single-byte instructions occur in a row, the disassembly resynchronizes at most after the maximum instruction length, which is 15 bytes. It is therefore important to investigate the probability for the transitions between instructions of different lengths.





14: Interleaving instructions: The graph on the left shows the instruction-length distribution for the different benchmark programs. Code interleaving is able only to accommodate up to 8 bytes within a masked instruction. As a result, the average number of interleave operations per block is limited (right).



15: Code overlapping results: Performance (left) for the three SpecCINT benchmarks under different overlapping factors $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}$.

Overall Performance.

Finally, we measure the performance impact of code overlapping. Figure 15 shows that performance of the tamper-hardened binary can be three times slower than the original binary, but often it is not necessary to overlap the whole binary. The results also show that overlapping provides some code compression. Normally, to improve tamper-resistance by a factor of *n*, a program must run at least *n* times, thus slowing down performance by a factor of *n*.

6. RELATED WORK

There exists a large body of related work on software protection. Two general directions involve software tamper-resistance and code obfuscation. The origin of most related work is in Cohen's paper on operating-systems evolution [16]. This paper defines a set of mutations on how programs can change to make reverse-engineering harder, implying that obfuscation must take place iteratively. As compared to our work, however, program evolution refreshes systems dynamically as a defense against attacks. We apply all transformations instantly.

Program evolution is also strongly related to protection against outside attacks. The general idea is to transform programs arbitrarily into equivalent code with the purpose of decreasing the likelihood that there are similar exploits [22]. Side-channel attacks fall into the same category. It is often easier to derive program properties not by reverse engineering code, but by injecting faults or measuring execution times [30, 11, 38, 41, 28, 10]. One approach to protect against these simple attacks is to inject regular checks into the code [6].

Software guards are intended to accomplish software tamper-resistance [13]. A guard is injected into code and observes program state at runtime. The idea is that different guards create a web of trust and watch one another. An adversary has to attack multiple guards at the same time to avoid detection. A related but different approach is the verification kernel [47, 8, 26]. Here the program consists of many small kernels that are encrypted individually. When control passes to a kernel, the program verifies the checksum of the kernel, decrypts it, and executes it. The program distributes the checksum across multiple kernels to avoid a single point of failure. Oblivious hashing is a different approach for computing fast checksums [14]. A program computes checksums on the fly and compares the checksum values at various points against expected checksums to ensure that the execution was done properly. Other techniques include timing measurements of architectural properties to ensure tamper-resistance of code, but these often work only on limited platforms like embedded systems [39].

Software protection is basically a bootstrapping problem from the operating system and the hardware architecture on which the applications are running. In an open system like Linux or Windows, little protection is offered by the operating system, and any type of code can execute. Recent research has focused on additional protection from the operating system through attestation and memory curtaining [21, 5, 23, 27]. The idea is that the platform architecture verifies software before execution and then shields the processes' memory not only from one another through virtual memory, but also from a kernel that is controlled by an adversary. All of these approaches have a small trusted operating-system kernel that is verifiable. There are also several hardware-only approaches to tamper-resistance. They focus either on hiding a secret in a distributed system [48, 33] or on implementing execute-only memory [34, 43].

Code obfuscation itself focuses on hiding program properties from adversaries who reverse-engineer the code. As mentioned earlier, there are tools that protect purely against static analysis using opaque predicates and control-flow-graph flattening [45, 18]. In CFG flattening, the CFG becomes one main loop with a case-statement, but because of opaque predicates, it is difficult to determine how the single basic blocks are interconnected. Another line of work is white-boxing block-cipher implementations. It would be useful to have a fast public-key implementation if it were possible to obfuscate a secret key in software [15, 12, 1]. There are also a number of ad-hoc obfuscation methods (e.g., [17, 2]). As reported by hacker essays, one anti-disassembly scheme was implemented in the Macrovision SafeDisc protection scheme [3], but this was easily deobfuscated by custom tools and IDA scripts.

In theory, little is known about code obfuscation, but the main reason is the difficulty of defining the notion. Informally, a basic definition involves "source code" that provides little or no additional information over black-box access (i.e., observing only input-output behavior of a program). In the random-oracle model, such general obfuscation is impossible for Turing machines [9]. An active area of research is obfuscation of point functions (e.g., Unix password checks), where a precomputed hash is compared to an input hash. Using the random-oracle model, it is possible to obfuscate point functions [36]. In a weakened model based on probabilistic hash functions, obfuscation of point functions is also possible [46]. Point functions are not only useful for hiding passwords, but also for protecting privacy in databases [37].

7. CONCLUSIONS

In this work, we show how to implement a new approach to software tamper-resistance that does not require additional hardware. We use the concept of code overlapping to implement oblivious hashing in native code on commodity architectures. This improves tamper-resistance of existing binaries with reasonable performance impact. Our methods also apply to any variable-length byte-codes that allow control transfer inside instructions, facilitating code-byte hashes without explicitly reading code bytes.

We demonstrate that existing applications for anti-disassembly cannot be used extensively without high performance impact because of the Kruskal-Count phenomenon. The Kruskal Count explains why disassembly resynchronizes quickly.

To address this issue, we propose code interleaving in combination with code outlining. This is also a novel approach to protect against checksumming attacks devised to defeat self-checking code [47]. Beyond anti-disassembly, interleaved machine instructions complicate an adversary's attempts to tamper with code or replace complete code patterns, since changing one instruction may inadvertently alter others. Via code outlining, we also use a similar principle at a semantic level; an attacker who patches an outlined code fragment may experience difficulties if multiple distinct code paths execute the same fragment for different semantic purposes. Like oblivious hashing, our methods are intended not to be "uncrackable," but to serve as an element of a comprehensive software-protection solution.

8. REFERENCES

[1] Cloakware. `http://www.cloakware.com`.

[2] Dotfuscator. `http://www.preemptive.com/products/dotfuscator`.

[3] Safedisc. `http://www.macrovision.com/products/safedisc`.

[4] SpecCPU benchmark. `http://www.spec.org`.

[5] Trusted Computing Platform Alliance. `http://www.trustedpc.org`.

[6] M. Abadi, M. Badiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS 2005*.

[7] B. Anckaert, M. H. Jakubowski, and R. Venkatesan. Proteus: Virtualization for diversified tamper-resistance. In *2006 ACM DRM Workshop*, 2006.

[8] D. Aucsmith. Tamper-resistant software: An implementation. In *Information Hiding*, 1996.

[9] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO 2001*.

[10] D. Boneh and D. Brumley. Remote timing attacks are practical. In *Usenix Security 2003*.

[11] D. Boneh and R. Lipton. On the importance of checking cryptographic protocols for faults. In *Eurocrypt 1997*.

[12] M. Cary, M. Jakubowski, and R. Venkatesan. Iterated obfuscation for white-boxing AES-like cipers (not published).

[13] H. Chang and M. J. Atallah. Processing software code by guards. In *ACM DRM 2001*.

[14] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding 2002*.

[15] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. A white-box DES implementation for DRM applications. In *ACM DRM 2002*.

[16] F. Cohen. Operating system protection through program evolution. `http://all.net/books/IP/evolve.html`.

[17] C. Collberg, G. Myles, and A. Huntwork. Sandmark - a tool for software protection research. *IEEE Transactions on Software Engineering*, 28(8), 2004.

[18] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[19] S. Debray and C. Linn. Obfuscation of executable code to improve resistance to static disassembly. In *CCS 2003*.

[20] N. Dedic, M. H. Jakubowski, and R. Venkatesan. A graph game model for software tamper protection. In *Information Hiding*, 2007.

[21] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, 2003.

[22] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS 1997*.

[23] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine based platform for trusted computing. In *SOSP 2003*.

[24] W. H. Gates. Personal communication.

[25] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society.

[26] B. Horne, C. S. Lesley, R. Matheson, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *ACM DRM 2001*.

[27] G. Hunt, J. Larus, D. Tarditi, and T. Wobber. Broad new OS research. In *HotOS 2005*.

[28] M. Jacob, D. Boneh, and E. Felten. Attacking an obfuscated cipher by injecting faults. In *ACM DRM 2002*.

[29] M. H. Jakubowski and R. Venkatesan. Protecting digital goods using oblivious checking, US Patent No. 7,080,257, filed on Aug. 30, 2000, granted on July 18, 2006.

[30] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Crypto 1996*.

[31] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Usenix 2004*.

[32] J. C. Lagarias, E. Rains, and R. J. Vanderbei. The Kruskal Count. `http://xxx.lanl.gov/math.PR/0110143`.

[33] R. Lee, P. Kwan, P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA 2005*.

[34] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy

and tamper-resistant software. In *ASPLOS-IX*.

[35] C.-L. Lin, H.-Y. Chen, and T.-W. Hou. Tamper-proofing of Java programs by oblivious hashing. In *CTHCP 2005: 11th Workshop on Compiler Techniques for High-Performance Computing*.

[36] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *Eurocrypt 2004*.

[37] A. Narayanan and V. Shmatikov. Obfuscated databases and group privacy. In *CCS 2005*.

[38] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*.

[39] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP 2005*.

[40] U. Shankar, M. Chew, and J. D. Tygar. Side effect are not sufficient to authenticate software. In *Usenix Security 2004*.

[41] D. Song, D. Wagner, and X. Tian. Timing attacks of keystrokes and timing attacks on SSH. In *Usenix Security 2001*.

[42] A. Srivastava, A. Edwards, and H. Vo. Vulcan – binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, MSR, 2001.

[43] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *ICS 2003*.

[44] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms, 2001.

[45] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, Dec. 2000.

[46] H. Wee. On obfuscating point functions. In *STOC 2005*.

[47] G. Wurster, P. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper-resistance. In *IEEE Security and Privacy 2005*.

[48] X. Zhuang, T. Zhang, and S. Pande. Hide: An infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS-XI*.