Identifying and Understanding Self-Checksumming Defenses in Software

Jing Qiu
Harbin Institute of Technology
Harbin, China
topmint@hit.edu.cn

Babak Yadegari
The University of Arizona
Tucson, USA
babaky@cs.arizona.edu

Brian Johannesmeyer
The University of Arizona
Tucson, USA
bjohannesmeyer
@cs.arizona.edu

Saumya Debray
The University of Arizona
Tucson, USA
debray@cs.arizona.edu

Xiaohong Su
Harbin Institute of Technology
Harbin, China
sxh@hit.edu.cn

ABSTRACT

Software self-checksumming is widely used as an anti-tampering mechanism for protecting intellectual property and deterring piracy. This makes it important to understand the strengths and weaknesses of various approaches to self-checksumming. This paper describes a dynamic information-flow-based attack that aims to identify and understand self-checksumming behavior in software. Our approach is applicable to a wide class of self-chesumming defenses and the information obtained can be used to determine how the checksumming defenses may be bypassed. Experiments using a prototype implementation of our ideas indicate that our approach can successfully identify self-checksumming behavior in (our implementations of) proposals from the research literature.

1. INTRODUCTION

Self-checksumming is widely used in software anti-tampering defenses [3, 5, 6, 8, 9, 14, 18]. The idea is to compute a hash value from the instructions of the program (or something closely related to those instructions) and ensure that the program continues to function correctly if and only if the computed hash has the expected value. This can be used to protect software against piracy, since any attempt to tamper with the code, e.g., to disable or remove a license check, will be detected during checksumming. This makes it important to understand the strengths and weaknesses of different approaches to self-checksumming.

This paper describes a dynamic information-flow-based attack that aims to identify and understand a large class of self-checksumming behavior in software. Our analysis provides a wide range of information about the checksumming code, such as: whether self-checksumming is being carried out and, if so, the location(s) of the code performing the checksumming; the origin of the checksum code (if it is created or modified dynamically); the checksum values computed; the locations of the code checking these checksum values; the mechanism by which the tamper-response is triggered (e.g., via a conditional jump, indirect jump, or through an unpacking key computed from the checksum); whether or not the instruction(s) triggering the tamper response are shared with any non-checksumming code; and so on. From an attacker's perspective, such information can provide a great deal of insight into the checksumming code and indicate how the self-checksumming can be bypassed or defeated. From the defender's perspective, such information can illuminate weaknesses in the self-checksumming code and possibly suggest remedies.

Our approach makes the following assumptions:

1. **Self-containedness.** The software performs its own integrity checking. This excludes systems, like Conqueror [14], that involve an external entity for verification.

2. **Observability.** The attacker has complete access to the host and is able to observe the program as it executes, including the instructions executed and the values of registers and memory.

Unlike the earlier work of Wurster *et al.* [22, 24], the work described here is a pure-software approach that does not rely on hardware assistance. To the best of our knowledge, this is the first such pure-software attack against self-checksumming systems. In terms of relative power, our approach does not seem directly comparable with that of Wurster *et al.*: on the one hand, our approach is unaffected by techniques, such as self-modifying code, that can defeat Wurster *et al.*'s attack [8]; on the other hand, programs that do not satisfy the assumptions listed above cannot be handled using our approach but may be susceptible to Wurster *et al.*'s attack.

The remainder of this paper is organized as follows. Section 2 provides background on self-checksumming; Section 3 describes our approach to detection of self-checksumming; Section 4 gives evaluation results for a prototype implementation of our ideas; Section 5 discusses limitations and future work; Section 6 discusses related work; and Section 7 concludes.

CODASPY'15, March 02 - 04 2015, San Antonio, TX, USA

http://dx.doi.org/10.1145/2699026.2699109.

2. BACKGROUND

Self-checksumming is widely used in anti-tampering mechanisms that aim to ensure that software that is going to be executed on an untrustworthy host has not been modified in unauthorized ways [3, 5, 6, 8, 9]. The idea is to compute a checksum or a hash value over appropriate portions of the program's code and use this value in the subsequent computation in such a way that the program executes as expected if and only if the hash value computed is the expected one. Compared to other proposals for software integrity protection, which require additional special-purpose hardware [21] or continuous connection with a remote "authentication server" [14], self-checksumming has the advantage of being implementable using commodity hardware and software. In particular, since self-checksumming does not require access to an authentication server, it is readily usable on mobile devices, such as smartphones and laptops, that may not always have network connectivity.

Horne *et al.* [9] divide this approach into two categories: *static* checksumming, which checks the static code of the program prior to execution; and *dynamic* checksumming, which checks the software as it executes.

2.1 Static Self-Checksumming

Static self-checksumming verifies the integrity of the software once, generally at the beginning of execution, to ensure that the disk image of the software has not been tampered with. This is typically done by accessing the bytes of the program file (whose name is usually passed to the executing process as an argument by the operating system), either by reading the file into memory or by mapping the file into the process's address space. Once the contents of the executable file are available for access, some or all of the code can be checksummed in a straightforward way.

2.2 Dynamic Self-Checksumming

While static self-checksumming can detect changes made to the program executable, it cannot detect changes made to the memory image of the program's code during the course of its execution. Dynamic self-checksumming addresses this problem by periodically checking the memory image of the program as it executes. The process of dynamic self-checksumming can be thought of, conceptually, as consisting of three components: (1) *checksum code insertion*, which inserts the code for computing the checksum into the program (if necessary); (2) *checksum computation*; and (3) *verification and tamper response*, which checks whether the checksum value computed matches the expected value and responds appropriately if it does not.

In many cases, no separate insertion component is necessary because the code that performs the checksum computation is compiled as part of the program's code and does not have to be inserted separately. An alternative approach, which aims to make the checksum code harder to identify, is to unpack or import the checksum code into the program at unpredictable intervals and/or locations in memory.

The checksum may be computed either on the code that is actually executed, or on non-executable memory locations containing a packed version the code (see Figure 1). The checksum computation may, but need not, encompass all of a program's code. First, the checksum computation may be limited in scope to some specific sensitive code modules whose integrity has to be enforced. Second, the overall

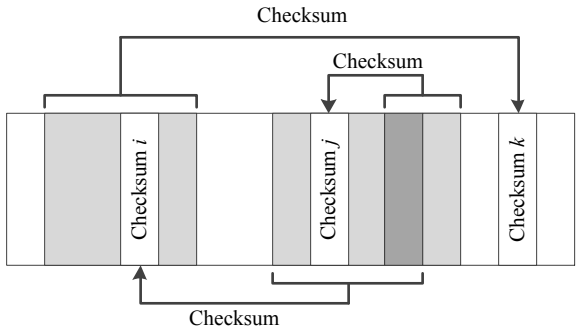

Figure 2: Program protected using multiple guards

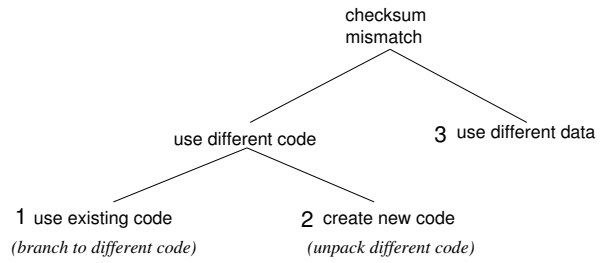

Figure 3: Tamper response: design choices

checksum computation may, in general, be carried out by a collection of distinct code snippets working together, with each snippet computing a checksum on some limited range of code. This makes possible a powerful self-checksumming model that uses a network of different checksumming routines that protect each other [6, 9]. Figure 2 shows an example of such approach. In this example the program has three checksumming routines each of which compute a checksum over the area of the code marked as grey and compares it with the pre-computed checksummed accordingly. As shown in Figure 2, checksum routines can have overlaps, meaning that they verify each others' integrity as well as the integrity of the code. To attack such a protection mechanism, the attacker has to detect all of the checksumming routines and disable them all at once; this is likely to be significantly more challenging than identifying and disabling a single guard.

The final step in self-checksumming is to check that the computed checksum matches the expected value and to activate a tamper response, where the program's execution behavior deviates from the normal, if it does not. Figure 3 illustrates the design choices for this step: in order for program behavior to deviate from normal, either the code or the data have to be different than for normal execution; and the execution of different code can be done either by branching to existing code, or by creating (unpacking) code that is different than what would normally have been created. Accordingly, the tamper response can be activated in one of three ways (the numbers in the list below correspond to those in Figure 3):

1. The program can branch to code that activates the tamper response. This can be done using either (*i*) straightforward compare-and-conditional-branch logic; or (*ii*) using the checksum value to compute the tar-

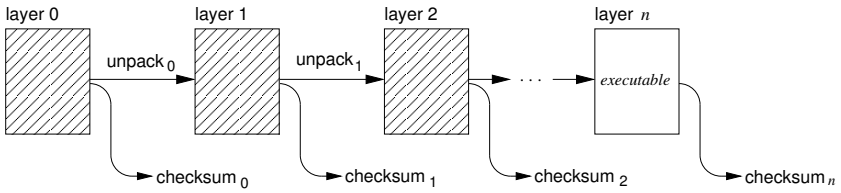

Figure 1: Checksumming combined with multiple layers of unpacking.

get of an indirect jump such that the correct target address is computed if and only if the checksum value computed is the expected one. For example, Tan *et al.* describe a scheme where a checksum mismatch causes control to be transferred to code that corrupts the program state by setting some pointers to NULL; the program then eventually crashes when this corrupted pointer is dereferenced [19].

2. The checksum value can be used to change the result of code unpacking: e.g., by using the checksum value to affect either (*i*) the value of the decryption key for some dynamically unpacked code [5,23]; and/or (*ii*) the value of one or more bytes that subsequently get unpacked and executed. In this case, an incorrect checksum value silently produces and executes incorrect/garbage code.

3. The checksum value can be incorporated into the logic of the computation in such a way that an incorrect checksum value causes the program to silently produce incorrect output. The following simple example illustrates this approach: the variable `p` is initialized to the correct value, 1, if and only if the computed checksum is equal to the expected value of `0x1234`.

```
int factorial(int n) {
    int cksum = compute_checksum();
    // expected checksum = 0x1234

    int p = 1 + (cksum ^ 0x1234);
    while (n > 0) {
        p *= n--;
    }
    return p;
}
```

   Since programs interact with their execution environments through system calls, this approach requires that an incorrect checksum should affect the argument(s) to some output system call.

The discussion in this paper focuses primarily on dynamic checksumming.

3. SELF-CHECKSUMMING DETECTION

3.1 An Overview of Our Approach

Intuitively, checksumming involves two kinds of computation:

1. computing a value from the contents of locations that either contain code (i.e., are executed) or are used to create code (e.g., through unpacking); and

2. using the value so computed (or a value derived from it) to affect the code the program executes and/or the output(s) it produces.

The key insight behind our approach is that both these computations can be identified using (different kinds of) taint propagation. Since many software protection tools use runtime code unpacking (e.g., Obsidium [1], Themida [16], VMProtect [2]), we do not rely on static analysis, which is unable to examine dynamically created code; instead, we use dynamic analysis.

Fig. 4 gives an overview of our approach. It consists of the following steps:

1. **Execution tracing**. The only input of our approach is an instruction trace of the target program. This can be done by using tools such as Intel Pin [13] or Ether [7].

2. **Backward taint analysis**. This step identifies memory locations that are either code (i.e., instructions or parts of instructions) or which are used to create code. We first walk through the trace and collect the addresses of executed memory locations. We then propagate taint in a backward direction, i.e., from uses to definitions, to identify locations that may have been used to create the executed code.

3. **Forward taint analysis**. This step identifies the flow of values computed from code locations to instructions that affect the code executed by the program. This is done by starting from the locations tainted in the backward taint analysis step and then propagating taint forwards (i.e., from definitions to uses).

4. **Checksumming detection**. This step identifies the checksum verification instructions. It looks for instructions I in the execution trace satisfying any of the following:

   (a) I is a control transfer instruction with one or more tainted operands; or

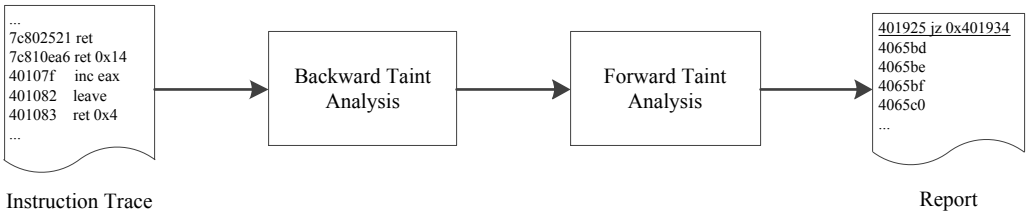

Figure 4: Overview of our approach

(b) I writes a tainted value to a location that is subsequently executed; or

(c) I passes a tainted value to an output system call.

Once the checksum verification instructions have been identified, further analysis can be performed, starting from these verification instructions, to obtain additional details about the checksumming code for the program.

The combination of backward and forward taint computation is necessary because it is possible to set up the checksum computation so that it considers, not the locations that are actually executed, but locations from which the instructions at those executed locations were created. For example, a piece of sensitive code—say, a license check or anti-analysis defense—may be stored in encrypted form in a memory region R, and decrypted as needed into some other memory region S from which it is executed; meanwhile the checksum computation can be applied to the memory region R, which is not itself executed. The backward taint analysis starts with the executed code in S, goes backward to taint R, then propagate this taint forward to the instruction(s) that perform checksum verification.

We use the following notation and terminology in the discussion that follows. Given an instruction I, $\mathsf{addr}(I)$ denotes the memory address of I; $\mathsf{length}(I)$ denotes its length in bytes; and $\mathsf{Read}(I)$ and $\mathsf{Write}(I)$ denote, respectively, the sets of locations read and written by I. An instruction I is tainted if $\mathsf{Read}(I)$ and/or $\mathsf{Write}(I)$ is tainted.

3.2 Taint Analysis

We perform byte-level taint analysis on instruction traces. Taint can be propagated in either a forward direction (i.e., the same direction as control flow) or backward (i.e., opposite to control flow). In forward taint propagation, the destination bytes of an instruction are tainted if and only if they are affected by the tainted bytes of the source; in backward taint propagation, source operand bytes are tainted if they affected the tainted bytes of the destination. Our current prototype handles taint propagation for the following instructions: data movement instructions, including string operations, push and pop instructions, and the `lea` instruction; arithmetic and logical instructions; shift and rotate instructions; and instructions that access the EFLAGS register. For instructions whose result does not depend on the source operands, e.g., "`xor eax, eax`" or "`sub eax, eax`", the result of the operation—in these examples, the contents of register `eax`—is marked as untainted.

Backward taint analysis is used to identify locations that are either executed or used to create/modify locations that are executed. Since this phase is used to identify locations that are used to create or modify code at runtime, it only propagates taint through data locations but not through the condition codes (the EFLAGS register) of the underlying x86 platform. Taint is propagated backward from a location ℓ at some point in the execution trace if and only if ℓ is live at that point. This means that when propagating taint across a data movement instruction I whose destination operand is tainted, at the point immediately before I the source operand of I will be tainted and the destination will be marked as untainted (since the destination is not live at the point right before I).

In forward taint analysis, taint is propagated through both data locations and condition code flags. For the instructions that use flags of EFLAGS, such as ADC (Add with Carry), the flag(s) read by the instruction are considered as source operands: if a flag read by the instruction is tainted, the destination will be tainted.

3.3 Identifying Self-Checksumming

Algorithm 1 gives a high level overview of the self-checksumming detection algorithm. The algorithm consists of three steps:

1. The first step is locating the source of the executed code using backward taint analysis on the program's execution trace (Algorithm 2). However, the locations in the source of the executed code will be excluded if they are not read or written by any other instruction (Algorithm 1 lines $4-6$).

2. Next we perform forward tainting on the execution trace, starting using the tainted locations from the backward taint phase as the taint source, to identify instructions with tainted operands that can affect the program's observable behavior (Algorithm 3). As part of this computation, as a "pre-forward-taint" step to accelerate the forward tainting step, we check whether any instruction reads or writes any tainted locations (Algorithm 3 lines $3-11$): if there are none, then the program does not do any self-checksumming and the algorithm exits. Otherwise, for each such instruction, we collect together the range of memory locations it checksums (Algorithm 3 lines $12-22$).

3. Finally, those instructions in this set for which the range of locations exceeds a (user-definable) threshold θ_{min} are identified as the checksum verification instructions (Algorithm 1 lines $9-13$).

In x86, a functions call is translated to a combination of `push` instructions and a `call` instruction. Such call instructions often do not read or write tainted memory locations. Thus, a call instruction that invokes an output system call with a tainted parameter will be regarded as a tainted instruction as well (Algorithm 3, lines $7-9$). So we can detect

Algorithm 1: Algorithm for detecting code checksummings. θ_{min} and θ_{max} are (user-definable) thresholds used to control the granularity of checksum reporting.

Input: T: Instruction trace $(I_0, I_1, \ldots, I_N)$
Output: R: Code self-checksum information

1 $R \longleftarrow \emptyset$;
`// Step 1: Backward taint analysis`
2 $M, AccessedM =$ `BackwardTaintAnalysis(`T`);`
3 $ExecutedMem \longleftarrow M$;
4 $M \longleftarrow M \cap AccessedM$;
5 **if** $M == \emptyset$ **then**
6 **return** R;
`// Step 2: Forward taint analysis`
7 $RS =$ `ForwardTaintAnalysis(`*T, M, ExecutedMem*`)`
`// Step 3: Summarize the result`
8 $Ignored \longleftarrow 0$;
9 **foreach** *<I, Locations> in RS* **do**
10 **if** *Locations.size()*$> \theta_{min}$ **then**
11 $R \longleftarrow R \cup \{< I, Locations >\}$;
12 **else**
13 $Ignored$ += *Locations*.size()
14 **if** $Ignored > \theta_{max}$ **then**
15 WARNING("no. of ignored locations exceeds θ_{max}");
16 **return** R;

the checksumming that its checksum flows to an output system call.

We use the threshold θ_{min} (step 3) to filter out false positives that occasionally arise in executables that use file compression to reduce their size. The issue is that if a file compressor finds the same byte sequence in multiple places within a file (some of which may be code while the other occurrences are unrelated data), this byte sequence is extracted out for compression purposes; during decompression (i.e., unpacking), this can cause bytes to be copied from an executed location to a non-executed location, in order to restore the multiple occurrences of that sequence. Our algorithm then marks the non-executed destination location as tainted during forward taint propagation, and any subsequent use of those locations, e.g., in a conditional branch, is flagged as being potentially a checksum verification. Our experience has been that the number of bytes involved in such coincidental matches is usually quite small. We therefore use the threshold θ_{min} to filter out matches involving only small regions of memory: in our prototype implementation, we set $\theta_{min} = 16$ bytes. We note that an attacker can try to defeat our algorithm using a collection of different checks, each of which covers fewer than θ bytes of memory but which collectively cover a significant amount of memory. To handle this, we add a global threshold θ_{max} on the number of memory bytes that can be ignored in this way. If the total memory size of the ignored regions exceeds θ_{max} the algorithm produces a warning to this effect (Algorithm 1 lines $14 - 15$).

As described in Figure 1, a program can have many layers of unpacking,[1] and self-checksumming can be performed at any point(s) in the unpacking sequence. Algorithm 1 can detect self-checksumming performed at any layer of unpacking. To see this, suppose at the ith level, function ϕ_i transforms data C_{i-1} to C_i, i.e. $C_i = \phi_i(C_{i-1})$. Suppose that there are n levels of unpacking and the resulting executable code is C_n. $CHECK(C_i)$ denotes a checksumming that verifies the integrity of C_i. The execution trace has the structure

$$
\begin{aligned}
&C_1 = \phi_1(C_0)\\
&\ldots\\
&C_2 = \phi_2(C_1)\\
&\ldots\\
&C_n = \phi_n(C_{n-1})\\
&\ldots\\
&C_n
\end{aligned}
$$

In the first step of the algorithm, all executed locations are collected and used as the taint source; this includes the instruction addresses in C_n as well as the code for all of the ϕ_i. Since ϕ_n reads C_{n-1}, the locations for C_{n-1} will become tainted. It is a straightforward induction to show that, proceeding in this way, all of the C_i as well as ϕ_i will be backward tainted; these locations will then be the taint source of the forward taint analysis. Now suppose that checksumming occurs before the kth unpacking step ($1 \leq k \leq n$), and the set of locations checksummed is $C'_{k-1} \subseteq C_{k-1}$.

Suppose that the checksummed locations C'_{k-1} originated in some set of locations C'_0 in the original packed representation. The backward taint analysis will mark C'_0 as tainted, and therefore the locations C'_{k-1} read by the checksumming code $CHECK(C_{k-1})$, as well as the values involved in the checksum computation itself, will be marked as tainted during the forward taint propagation from C'_0. As a result the values flowing into the checksum verification will be tainted and so the self-checksumming will be detected.

Algorithm 2: Backward taint analysis procedure

1 **Function** `BackwardTaint(`T`)`
2 $M \longleftarrow \emptyset, AccessedM \longleftarrow \emptyset$;
3 **foreach** *Instruction I in T* **do**
4 **for** $i \longleftarrow 0$ **to** $\mathsf{length}(I)$ **do**
5 $M \longleftarrow M \cup \{\mathsf{addr}(I) + i\}$;
6 $AccessedM \longleftarrow AccessedM \cup \mathsf{Read}(I) \cup \mathsf{Write}(I)$;
`// Backward taint`
7 **for** $i = N$ **to** 0 *Step -1* **do**
8 Backward taint $T[i]$ and update M accordingly;
9 **return** $M, AccessedM$

3.4 Understanding Self-Checksumming

Once the checksum verification instruction(s) have been identified, the taint information gathered from the forward taint propagation step can be used, possibly in conjunction with some additional analysis of the execution trace, to extract a variety of information about the self-checksumming protections deployed by the program under study. This information can be useful in guiding efforts to defeat or bypass the program's self-checksumming anti-tamper defenses. This section briefly describes some of the information that can be obtained in this way.

[1]Some software protection tools produce executables with dozens or hundreds of layers of runtime code unpacking [10].

Algorithm 3: Forward taint analysis procedure

```
1  Function ForwardTaintAnalysis(T, TaintedMem, ExecMem)
2      map<Instruction, set<Location> > RS;
3      M ⟵ TaintedMem, T′ ⟵ ∅, Calls ⟵ ∅;
4      foreach Instruction I in T do
5          if (Write(I) ∪ Read(I)) ∩ M ≠ ∅ then
6              T′.push_back(I);
7          else if I invokes an output system call C
8                  ∧ A parameter of C is tainted then
9              T′.push_back(I), Calls ⟵ Calls ∪ {I};
10         Forward taint I and update M accordingly;
11     if T′ == ∅ then return RS ;
12     foreach Location L in TaintedMem do
           // Forward taint L
13         M ⟵ {L};
14         foreach Instruction I in T′ do
15             if I ∉ Calls then
16                 Forward taint I and update M
                   accordingly;
17                 if (Write(I) ∪ Read(I)) ∩ M ≠ ∅ then
18                     if I is a control transfer instruction
19                       ∨ Write(I) ∩ ExecMem ≠ ∅ then
20                         RS[I].insert(L);
21             else if A parameter of the call I invokes is
                   tainted then
22                 RS[I].insert(L);
23     return RS
```

I. Checksum Computation.

To obtain the pieces of code that compute checksums, we can compute a backward dynamic slice [12] from the operands of each occurrence of a set of checksum verification instructions in the execution trace. There are two points to note here. First, the slicing algorithm has to take into account the unstructured nature of executable code [11]. Second, the control flow graph of the program may not be readily available: in this case, we use the execution trace to construct a control flow graph for the portions of the code that were executed. Since the program under consideration may use dynamically unpacked code (possibly with many layers of unpacking), the same memory address can contain different instructions at different points in the program, so an instruction cannot be identified by its memory address alone, but requires an additional parameter indicating how many times it has been modified. The reason we compute these slices at each dynamic occurrence of a checksum verification instruction is that it is possible for different checksum computation codes to share the same verification code. Note that sharing verification code in this way may not be a good idea since it can reduce the overall security of the system; our point here is simply that, even if the verification code is shared between many different checksum computations, our analysis can tease them apart. This approach can also tease apart different checksum computations even if their executions overlap (e.g., due to being run in different threads) and so are interleaved in the trace.[2]

The dynamic slice computation also provides information about the origin of the checksum computation code (see Section 2). Specifically, if any of the locations that are executed during the checksum computation are modified prior to execution (via memory writes), then the checksum code is created/unpacked dynamically. In this case, attempts to disable the checksum computation (e.g., by having it returns a precomputed checksum value, which is in fact available in the execution trace) may prefer to focus on the source of the unpacked code (which is also available in the dynamic slice) rather than the unpacked code itself.

Finally, the dynamic slice can be used to identify violations of the self-containedness assumption of Section 1. If any component of the checksum computation—either the checksum computation code or a seed for the checksum value—is obtained from an external source (e.g., a remote server), some components of the slice will be seen as originating from network reads; if the checksum computed by the program is communicated to an external entity for verification, then a tainted value will be seen as an argument to a network write. This can be used to understand the behavior of anti-tamper systems such as Conqueror [14].

II. Checksum Verification.

An attacker may try to bypass the self-checksumming defenses by altering the checksum verification code, e.g., by inserting an unconditional jump to the "normal execution" code. Knowing the conditions under which such an attack will or will not work can be helpful for guiding the attacker in selecting an attack. This can be done using information from the taint analysis. This simple attack may not work under the following conditions:

1. the tamper response is invoked using a control transfer instruction I, where some executions of I in the trace have a tainted operand (indicating a checksum computation) and other executions of I have no tainted operands (indicating a non-checksum computation); or

2. a forward-tainted value is written to a backward-tainted location (indicating that the value of the computed checksum is used to create or modify code that is subsequently executed), or loaded as an argument to an output operation in the program (e.g., a `write` operation).

The situation described in the first of these conditions can arise in virtualization-obfuscated code (e.g., we observed it in code protected using Themida [16]). This kind of obfuscation embeds the program logic in the byte-code of a custom virtual machine; the executable code for the program consists of the emulator for this virual machine. The emulator uses a handler for each different operation of the virtual machine, e.g.:

```
handle_if_EQ:  /* if_EQ op1, op2, target */
  op1 = fetch_op1();
  op2 = fetch_op2();
  target = fetch_op3();
  ip = (op1 == op2)? target : (ip + 1);
  goto emulator_dispatch;
```

[2]In order to distinguish between instructions from different threads, the execution trace has to record a thread-id for each instruction.

Thus, the code fragment shown above will be executed whenever an "`if_EQ`" operation is encountered in the byte code, including for example the checksum verification (if the verification is done using an "`if_EQ`" operation). Thus, the conditional branch that performs the checksum verification (in which case the operands are tainted) is shared with other non-checksum-computations (in which case the operands are not tainted). Naively altering the operation of this code will therefore alter the behavior of all such conditional branches in the byte code, not just the checksum verification code.

3.5 Examples

This section illustrates our approach using three simple examples.

EXAMPLE 3.1. This example illustrates "standard self checksumming", which refers to programs that do not have any code unpacking or runtime code generation and which use explicit control transfers to branch to the tamper response.[3]

```
/* compute checksum */
checksum = 0;
for (i = 0; i < N; i++){
    checksum += buf[i];
}

/* verify checksum */
if (checksum != V) {
    // tamper response
}
```

(a) Self-checksumming code

```
checksum = 0
checksum += buf[0]
checksum += buf[1]
...
checksum += buf[N-1]
cmp checksum, V
jnz tamper_response
```

(b) Trace (fragment)

Suppose `buf` points to the executed code section being checksummed: then the locations `buf+`i are used as taint sources during backward taint propagation, and the locations `buf+0`, `buf+1`, ..., `buf+N-1` are therefore tainted during this phase. Forward tainting from the locations `buf+`i, through the fragment of the trace shown above, will then cause the location `checksum` to become tainted. This will then cause the condition code flags to be tainted after the instruction "`cmp checksum, V`". Since the condition code flags are used as inputs to the conditional branch instruction "`jnz tamper_response`", our analysis will identify this conditional branch as taking tainted inputs (Algorithm 3, line 17) and therefore flag it as a tamper response instruction.

EXAMPLE 3.2. This example considers the situation where a checksum for one piece of code is used as the decryption key for a second piece of code: any tampering with the first code region results in the wrong decryption key being used for the second code section, resulting in the generation of garbage code by the unpacker. This approach to self-checksumming-based anti-tampering has been proposed by Cappaert *et al.* [5] and Wang *et al.* [23].

Suppose that, in the code fragment shown below, `buf` points to executed code and `buf2` points to packed code that is unpacked using the checksum value for `buf` as the decryption key:

[3]The "tamper response" here may simply involve setting a flag to indicate a checksum mismatch, with further actions delayed to enhance stealth, e.g., as suggested by Tan *et al.* [19].

```
/* compute checksum */
checksum = 0;
for (i=0; i < N1; i++) {
    checksum += buf[i];
}

/* decrypt code */
for (i=0; i < N2; i++) {
    buf2 ^= checksum;
}

/* execute code */
...
```

(a) Self-checksumming code

```
checksum = 0
checksum += buf[0]
checksum += buf[1]
...
checksum += buf[N1-1]
buf2[0] ^= checksum
buf2[1] ^= checksum
...
buf2[N2-1] ^= checksum

...
jmp buf2
```

(b) Trace (fragment)

Since `buf` and `buf2` point to executed code locations, the memory regions `buf+0`, ..., `buf+N1-1` and `buf2+0`, ..., `buf2+N2-1` are tainted during backward taint propagation. During forward taint propagation, the taint on `buf+0`, ..., `buf+N1-1` causes `checksum` to become tainted. Since `buf2+0`, ..., `buf2+N2-1` are also tainted, this then causes our analysis detects that, in the assignments to `buf2`, tainted values are written to an executed location and therefore flags this code as a tamper response (Algorithm 3, line 18).

EXAMPLE 3.3. This example shows that a checksum can be used for initializing a variable. The variable is taken in a computation whose result is finally displayed to the screen.

```
/* compute checksum */
checksum = 0;
for (i=0; i < N; i++) {
    checksum += buf[i];
}

int p=1+checksum^0x1234;
while (n > 0) p *= n--;
printf("%d\n", p);
```

(a) Self-checksumming code

```
checksum = 0
checksum += buf[0]
checksum += buf[1]
...
checksum += buf[N1-1]
p = 1+ checksum^0x1234
p *= n
n -= 1
cmp n, 0
jnz loop
...
push p
...
call printf
```

(b) Trace (fragment)

Suppose `buf` points to an executed code section. Forward tainting `buf+`i, the system call "printf" will be identified as a tamper response instruction because its second parameter p is tainted (Algorithm 3, lines $20 - 21$).

3.6 Implementation Considerations

The space and time costs of processing large traces can potentially be a concern for offline dynamic analyses such as ours. Trace compression techniques can significantly mitigate the storage and I/O costs of processing large traces: e.g., Bhansali *et al.* [4] describe a trace compression scheme that results in roughly 0.5 bits of trace data per dynamic instruction instance.

Additionally, we use two optimization techniques to speed up taint propagation, which is at the heart of our analysis: (i) reducing the number of instructions that have to be examined for taint propagation; and (ii) reducing the overall cost of forward taint propagation using parallelism.

Eliminating irrelevant instructions.

In order to reduce the number of instructions processed during forward taint propagation, we use a pre-tainting phase to propagate taint information and identify a sub-trace of

the original trace that contains all the relevant instructions for forward taint propagation (Algorithm 3, lines $3 - 9$). We can prove that forward taint propagation from this sub-trace is equivalent to forward tainting from the entire original trace. This can produce significant improvements in performance because the sub-trace obtained from this pre-tainting phase is usually significantly smaller than the original trace. As an example of the performance improvements obtained, for Media Player Classic the total time drops from 480 secs to 240 secs; while for the 50-guards program the total processing time improves by more than fourfold, going from 6,282 secs to 1,352 secs.[4]

Second, to avoid processing irrelevant instructions, we only trace instructions executed after the target program reaches its entry point. This avoids tracing and recording process startup code.

Exploiting concurrency.

We use multi-threading to parallelize forward taint propagation step. Since the trace is a read-only input for forward taint propagation, it can be processed in parallel without locking overheads. We partition the locations tainted during the backward-taint step into a fixed number of subsets and for each subset create a thread to perform forward tainting.

4. EVALUATION

4.1 Setup

This section discusses our experiences with using a prototype tool we developed to evaluate our ideas. Execution tracing is carried out using an Intel Pin tool [13]. The data presented here were obtained on a 2.67GHz Intel Xeon E5640 processor (12 MB L1 cache) with 96 GB of main memory running Ubuntu 12.04. Our tool was run with 16 parallel threads. The threshold θ_{min} and θ_{max} in Algorithm 1 are set to 16 and 512, respectively.

We used two sets of test programs to evaluate our tool. In each case, the test program we used is an MD5 computation program obtained from `http://people.csail.mit.edu/rivest/Md5.c`, executed with a text file of Abraham Lincoln's Gettysburg Address as input.

- Group 1 consists of seven widely used open source programs: Media Player Classic 1.7.6; Notepad++ 6.6.7; FileZilla 3.7.0; WinMerge 2.14.0; DOSBox 0.74; VLC 2.0.5; and 7-Zip 9.20. We used the source code for these programs to verify that none of them were performing any self-checksumming. This was used as a baseline to check that there were no false positives reported by our tool.

- Group 2 consists of a set of programs we wrote to implement advanced self-checksumming schemes for which we could not find any third-party tools.[5] In each case, we modified the MD5 program mentioned above to incorporate the self-checksumming mechanism. The checksumming schemes we tested were as follows:

  (i) Programs with multiple self-checksumming guards, as described by Chang *et al.* [6] and Horne *et al.* [9]. The objective was to test whether our approach can correctly identify multiple overlapping guards checking each other. We implemented and tested programs with one, four, ten, and fifty guards; in the results given below, we refer to these as *1-guard*, *4-guards*, *10-guards*, and *50-guards* respectively.

  (ii) Self-checksumming programs that use the value of the checksum as a code decryption key, as described by Cappaert *et al.* [5] and Wang *et al.* [23]. The objective was to test whether our approach can detect self-checksumming schemes where the checksum verification and tamper response step uses dynamic code modification instead of an explicit control transfer. In the results below, we refer to this program as *decrypt-key*.

  (iii) Checksumming combined with runtime code unpacking, as illustrated in Figure 1. The objective was to test whether our approach could detect self-checksumming when the locations being checksummed are not themselves executed, but are used to create code that is executed. We tested a program with 100 layers of unpacking, with checksumming carried out after each even-numbered unpacking layer for a total of 50 different checksum computations. In the results given below, refer to this program as *100-layers*.

  (iv) The checksum is used for generating a MD5 initialization constant. The MD5 value is output to the screen by $printf()$. In the results given below, we refer to this program as *chksum-md5*.

  These programs tested the precision and recall of our approach. The source code for these programs, as well as the executables obtained from them that we used in our tests, are available at `http://www.cs.arizona.edu/projects/lynx/Samples/Self-checksumming/`.

We validated the results obtained from our analysis as follows.

- For programs in Group 1, we compile their source code and generate debug information files. The debug information is a representation of the relationship between the executable program and the original source code. With these debug information, we validate the result of programs in Group 1 using a debugger to monitor the execution of each program.

- For Group 2, we instrumented the programs we constructed to report, at runtime, each address range that was checksummed each time a checksum was computed. This was then compared with the results reported by our tool.

[4]These timings refer to single-threaded execution time. The performance data given in Section 4 refer to multi-threaded execution times.

[5]In order to evaluate our algorithm on state-of-the-art commercial software anti-tampering systems, we also approached two commercial vendors who market anti-tampering systems that are based on peer-reviewed research publications on software self-checksumming. The vendors declined to provide access to their protection tools; one vendor cited concerns about the potential for adverse publicity resulting from our work.

4.2 Evaluation Results

The result of the evaluation is given in Table 1. The "Number of Taint Sources" column gives the total number of locations tainted during backward tainting; the "No. of Tainted Instructions" column gives the total number of instructions tainted during forward tainting. We did not find any false positives in the programs in Groups 1 and 2.

4.2.1 Precision of Analysis

Group 1.

We have checked the source code of programs in Group 1, and no checksumming is found in the source code. Our prototype tool does not find any code checksumming as expected. In the processing of most programs except Media Player Classic (MPC), our approach exit early after backward taint analysis because no instruction reads or writes the executed locations. MPC employs a third party library to hook system APIs. It writes an unconditional jump instruction at the head of the hooked API. The source of the bytes written is discovered by the backward taint analysis. But no checksumming is found in MPC as expected.

Group 2.

In Group 2, for programs with multiple guards, our approach successfully identifies all designed checksummings. However, the code coverage issues of dynamic analysis manifest themselves (see Section 5): for a specific input, not all of the protected code is executed at runtime, and since our approach starts the analysis with the set of executed locations, not all checksummed locations (i.e., the set of protected memory locations associated with each checksum verification instruction) are identified by our approach. Thus, for some of the checksum guards our tool reports a smaller range of checksummed locations than is in fact the case because some checksummed locations were not executed.

In the program with 100 layers unpacking, all checksummings are identified by our approach. Our tool reports that these checksummings have the same protected code range. That is because in the source code of this program, the checksummed code is transformed and checksummed in one layer, and then it is passed into next unpacking layer.

Our approach also successfully identifies the checksumming in the program that the computed checksum is used as a decryption key. The code of the checksumming is as follows.

```
checksum = compute_checksum(CODE);
for(i = 0; i < size; i++)
    CODE[i] -= checksum;
```

Instructions "`CODE[`i`] -= checksum`" and the range of `CODE` are reported by our tool.

The checksumming in "chksum-md5" is identified as well. The checksum is used for generating a MD5 initialization constant. If the protected code is tampered, the computed MD5 value is incorrect. The code is as followings.

```
int cksum = compute_checksum(CODE);
...
//This value should be 0x67452301;
mdContext->buf[0] = chksum + 0x6740E9CB;
...
printf("%s", md5_str);
...
```

The instruction "`call printf`" and the code range checksummed are reported by our tool.

The result of this group indicates that no matter how a checksum is used in an execution, the activity that computing a checksum over code will always be discovered by our approach.

4.2.2 Effects of Performance Optimization

Suppose there are L_1 instructions in an instruction trace and after backward taint analysis, there are L_2 locations; that forward tainting L_2 locations produce L_3 tainted instructions; and the implementation uses N threads. Define the *workload* of our approach be the total number of instructions processed. The number of instructions processed in the backward taint propagation phase is $2 * L_1$, while the total number of instructions processed during forward taint propagation is $L_2 * L_3$ without multi-threading and $L_2 * L_3/N$ with multi-threading. Thus, the unoptimized and optimized workloads are given by $2*L_1+L_2*L_3$ and $2*L_1+L_2*L_3/N$ respectively. The effect of multi-threading is to significantly reduce the workload for the forward taint propagation phase.

Table 2 shows that the optimization sharply decreases the work load. In Group 1, most programs' analysis exit early because we found that no instruction reads or writes executed locations. Thus, the optimization is not worked and the work load is $2 * L_1$. In the analysis of the reset programs, the optimized approach only processes average 2.92% instructions of the un-optimized approach while obtaining the same result. It indicates that excluding irrelevant instructions and the parallel processing technology make our approach more practical to real world binary analysis. When dealing with the trace file of a large scale program, just simply split the work load and assign them to more processors. The more processors join, the less time will take.

Table 2: Work load of the approach without and with optimizations. The work load is defined as the number of processed instructions.

Program	Without ($\times 10^6$)	With ($\times 10^6$)	With/Without (%)
1-guard	609	13	2.16
4-guards	727	16	2.20
10-guards	1,847	45	2.41
50-guards	13,505	540	4.00
100-layers	3,740	161	4.31
decrpt-key	392	11	2.80
chksum-md5	693	18	2.58

4.3 A Case Study

As a case study of our ideas, we applied our prototype tool to a well-known and widely used Internet communication application. Our initial experiments use an execution trace up to the point where the application's splash screen appears (approx. 62.6 million instructions).

After the program begins execution, it unpacks some code (this is done *in situ* rather than generating code into some fresh memory region) using the following unpacking logic:

Table 1: Evaluation result

Program	Trace Size		No. of Tainted	No. of	No. of Guards		Analysis Time
	Mbytes	Instructions	Instructions	Taint Sources	Found	Ground Truth	(sec)
1-guard	11	179,339	60,298	3,394	1	1	5
4-guards	13	209,926	71,885	3,641	4	4	6
10-guards	22	375,563	142,483	4,916	10	10	16
50-guards	160	2,780,558	1,760,264	4,855	50	50	237
100-layers	92	1,600,653	1,083,861	2,335	50	50	53
decrpt-key	9	166,581	72,422	2,352	1	1	4
chksum-md5	17	297,068	118,526	2,353	1	1	5

```
SEED = 0x135a936d;
unsigned char* CODE = &SOME_CODE_SECN; // 0x5bc1f0
for(int i = 0; i < 0x16160; ++i) {
    if (i == 0x15f08) i += 0x28; // Skip 0x28 bytes
    CODE[i] ^= SEED & 0xFF;
    rol(SEED, 3);  // rotate left
}
```

Somewhat later in the execution the application executes self-checksumming code that is part of the code unpacked earlier. The checksum computation and validation code has the following logic:

```
unsigned int chksum = 0x0A9C35B72;
unsigned char* CODE = &SOME_CODE_SECN; // 0x5bc1f0
for (i = 0; i < 0x16160; i++) {
    chksum = (chksum >> 4) + CODE[i];
    chksum2 = chksum & 0x0f000000;
    if (chksum2 != 0) {
        chksum ^= (chksum2 << 0x18);
    }
    chksum &= ~chksum2;
}

f = BIT_MASK ^ chksum;
...
call f;     // validation!
...
return;
```

The validation code in this case does not explicitly compare the computed checksum against an expected value, but rather uses it to compute the target of an indirect call (indicated above by the comment '`validation!`'): an incorrect checksum value simply transfers control to the wrong address and results in an incorrect execution.

The `for`-loop that computes the checksum in the code shown above contains an `if`-statement that checks whether the variable `checksum2` is nonzero. Since this is a conditional branch that tests the value of a checksum value, our tool flags this as a possible validation check. Since this `if`-statement is executed each time around the loop, our tool reports a series of different code regions $C_0, C_1, \ldots, C_i, \ldots$ that are "checked" by this test, such that each C_i is a proper prefix of each C_j for $j > i$. It would be straightforward to modify our tool to detect these overlapped regions and collapse them into a single reported region, however this is a reporting issue orthogonal to that of detecting and identifying self-checksumming.

We are currently analyzing a longer trace of this application (approx. 6.6 billion instructions long) and hope to report the results at the conference presentation.

5. DISCUSSION AND FUTURE WORK

While the dynamic analysis-based approach we use has the advantage of being able to deal transparently with anti-analysis defenses such as runtime code self-modification, it has the disadvantage of limited code coverage. The code coverage problem can be mitigated using multi-path exploration techniques [15].

A second problem with offline dynamic analysis is the potential for large trace files, which can incur significant storage and processing costs. This issue can be mitigated using trace compression techniques, e.g., Bhansali *et al.* [4] describe a trace compression scheme that results in roughly 0.5 bits of trace data per dynamic instruction instance.

Finally, while the offline analysis used by our approach can identify self-checksumming and provide a great deal of information about the specific defenses being used by a given program, it does not automatically translate into a straightforward way to disable the checksumming. In the experimental evaluation of our prototype, we manually validate the correctness of detection results. This can be tricky if the checksumming code is created dynamically at unpredictable locations in memory.

Our current prototype implementation does not currently incorporate trace compression or multi-path exploration: we plan to do so as part of future work to improve execution performance and code coverage. We also plan to explore online analysis algorithms and to extend our analysis to other kinds of anti-analysis defenses such as timing-based tracing detection.

The discussion in this paper focuses on dynamic checksumming and does not consider static checksumming. It is not difficult, in principle, to adapt our approach to handle static checksumming: all we need to do is to keep track of library/system calls to identify any access the program file, either through explicit file I/O or by mapping the file into the process's virtual address space. Once this has been done, it is not too difficult to parse the file structure and identify operations that read from locations within the file that correspond to code. Incorporation of such logic into our prototype implementation is the subject of future work.

6. RELATED WORK

There is a considerable body of work on anti-tampering defenses for software. Much of this work focuses on self-contained defenses based on code self-checksumming that meet our assumptions as described in Section 1. Aucsmith [3] introduced an implementation of a self-checking mechanism based on multiple self-modifying and self-decrypting code blocks that check the validity of the code as it is running. Chang *et al.* [6] and Horne *et al.* [9] discuss self-checksumming systems that use a network of guards such

that each guard is protected by multiple other guards. Disabling this kind of protection requires all the guards be disabled at once which makes it non-trivial for the attacker. Tsang *et al.* use a large number of small size protectors capable of giving non-pre-programmed tamper responses, and use multiple versions of a function for creating non-deterministic execution [20]. Wang *et al.* [23] and Cappaert *et al.* [5] propose the use of checksum values as a key to decrypt the encrypted code that is going to be executed, such that any tampering causes the code to be decrypted with the wrong key and silently produces incorrect code. Tan *et al.* discuss stealthy tamper response techniques that make it difficult for an attacker to tie the tamper response back to the checksum verification code [19]. By and large these works assume a threat model based on pure static analysis; their self-checksumming defenses can be clearly identified by our dynamic analysis-based approach. There has also been some work on anti-tampering defenses where the checksum verification is performed by an external verifier on a trusted remote server [14, 18]. These approaches do not use self-contained self-checksumming defenses, as discussed in Section 1, and so fall outside the scope of this work.

The only other work we know of that looks at attacks on self-checksumming code is that of Wurster *et al.* [22, 24], who exploit an assumption underlying self-checksumming approaches that the same byte values will be retrieved from a virtual memory address range regardless of whether it is retrieved as code or data. They show that an adversary hardware assisted techniques to violate this assumption and bypass the self-checksumming defense. Giffin *et al.* show that self-modifying code can be used to detect this attack [8]. Unlike Wurster *et al.*'s attack, the work we describe is a pure-software approach that does not rely on hardware assistance. To the best of our knowledge, this is the first such pure-software attack against self-checksumming systems. In terms of relative power, our approach does not seem directly comparable with that of Wurster *et al.*: on the one hand, our approach is unaffected by techniques, such as self-modifying code, that can defeat Wurster *et al.*'s attack [8]; on the other hand, programs that do not satisfy our self-containedness assumption (see Section 1) cannot be handled using our approach but may be susceptible to Wurster *et al.*'s attack.

There is a wide body of literature on various forms of taint analysis and their applications to software analysis, e.g., see [17]. To the best of our knowledge none of these works apply taint analysis to the problem of detecting or understanding self-checksumming.

Wang *et al.* proposed a fuzzing tool for software vulnerabilities detection. They use taint analysis to detect checksumming on the input data of a program. The checksumming is similar to ours but there two big differences. First, they identify data checksumming while we focus on code checksumming. Second, they identify the hot bytes which are checksummed bytes in the input data such that only the checksumming with classic verifier can be identified by their approach. Instead, our approach focus the identification of checksumming verifiers such that more kinds of checksumming can be identified by our approach.

7. CONCLUSION

This paper describes an information-flow-based attack against software anti-tampering defenses based on self-checksumming; to the best of our knowledge it is the first pure-software attack against such defenses. Our approach is based on dynamic analysis: it uses backward taint propagation to identify memory locations that are either executed or which are used to compute the values of locations that are executed, followed by a forward taint propagation that identifies checksum computations on the code. Experiments using a prototype implementation show that our approach can effectively identify a wide range of self-checksumming behaviors, including a number of proposals for self-checksumming that described in the research literature.

Acknowledgments

This research was supported in part by the Air Force Office of Scientific Research (AFOSR) under grant no. FA9550-11-1-0191; the National Science Foundation (NSF) under grants CNS-1115829, CNS-1145913, III-1318343, and CNS-1318955; and the US Department of Defense under grant no. FA2386-14-1-3016. The work of Jing Qiu was supported by National Natural Science Foundation of China (NSFC) 61173021. The opinions, findings, and conclusions expressed in this paper are solely those of the authors and do not necessarily reflect the views of AFOSR, NSF or NSFC.

8. REFERENCES

[1] Obsidium software protection system. `http://www.obsidium.de/show/home/en`.

[2] VMProtect – New-generation software protection. `http://www.vmprotect.ru/`.

[3] D. Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333. Springer, 1996.

[4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, 2006.

[5] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere. Towards tamper resistant code encryption: Practice and experience. In *Information Security Practice and Experience*, pages 86–100. Springer, 2008.

[6] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[8] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[9] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and privacy in digital rights management*, pages 141–159. Springer, 2002.

[10] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*, Nov. 2007.

[11] B. Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, Jan. 1997.

[12] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with

dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[14] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.

[15] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[16] Oreans Technologies. Themida: Advanced windows software protection system, Sept. 2008. `http://www.oreans.com/themida.php`.

[17] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[18] 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. *ACM SIGOPS Operating Systems Review*, 39(5):1–16, 2005.

[19] G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In *Information Hiding*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2006.

[20] H.-C. Tsang, M.-C. Lee, and C.-M. Pun. A robust anti-tamper protection scheme. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 109–118. IEEE, 2011.

[21] J. Tygar and B. S. Yee. Dyad: A system for using physically secure coprocessors. 1991.

[22] P. C. Van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):82–92, 2005.

[23] P. Wang, S. Kim, and K. Kim. Tamper resistant software through dynamic integrity checking. In *Proc. 2005 Symposium on Cryptography and Information Security (SCIS2005)*, Jan. 2005.

[24] G. Wurster, P. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.