Dynamic Self-Checking Techniques for Improved Tamper Resistance

Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan

STAR Lab, InterTrust Technologies
4751 Partick Henry Dr.
Santa Clara, CA 95054
{bhorne,lrm,casey,ret}@intertrust.com

Abstract. We describe a software self-checking mechanism designed to improve the tamper resistance of large programs. The mechanism consists of a number of *testers* that redundantly test for changes in the executable code as it is running and report modifications. The mechanism is built to be compatible with copy-specific static watermarking and other tamper-resistance techniques. The mechanism includes several innovations to make it stealthy and more robust.

1 Introduction

There are many situations in which it is desirable to protect a piece of software from malicious tampering once it gets distributed to a user community. Examples include time-limited evaluation copies of software, password-protected access to unencrypted software, certain kinds of e-commerce systems, and software that enforces rights to access copyrighted content.

Tamper resistance is the art and science of protecting software or hardware from unauthorized modification, distribution, and misuse. Although hard to characterize or measure, effective protection appears to require a set of tamper resistance techniques working together to confound an adversary.

One important technique is *self-checking* (also called self-validation or integrity checking), in which a program, while running, checks itself to verify that it has not been modified. We distinguish between *static* self-checking, in which the program checks its integrity only once, during start-up, and *dynamic* self-checking, in which the program repeatedly verifies its integrity as it is running.

Self-checking alone is not sufficient to robustly protect software. The level of protection from tampering can be improved by using techniques that thwart reverse engineering, such as customization and obfuscation, techniques that thwart debuggers and emulators, and methods for marking or identifying code, such as watermarking or fingerprinting. These techniques reinforce each other, making the whole protection mechanism much greater than the sum of its parts.

In this paper we describe the design and implementation of a dynamic self-checking mechanism that substantially raises the level of tamper-resistance protection against an adversary with static analysis tools and knowledge of our algorithm and most details of our implementation. Our threat model is described

in detail in Section 3. Our overall goal is to protect client-side software running on a potentially hostile host.

We begin in Section 2 with a brief discussion of related work. In Section 3 we address our threat model and the design objectives we used to create techniques to oppose these threats. Section 4 presents an overview of our self-checking mechanism and its components. Section 5 describes the design, performance and placement of the testing mechanism. Section 6 discusses the design and interconnection of the tested intervals of code. Finally, Section 7 concludes with a summary and a brief discussion of directions for future improvements.

The authors of this document were primarily responsible for the design and implementation of the self-checking technology. Throughout its evolution, however, many important contributions came from others, including Ann Cowan, Chacko George, Jim Horning, Greg Humphreys, Mike MacKay, John McGinty, Umesh Maheshwari, Susan Owicki, Olin Sibert, Oscar Steele, Andrew Wright, and Lance Zaklan.

2 Related Work

There has been a significant amount of work done on the problem of executing untrusted code on a trusted host computer [MWCG99,Nec98,NL96]. The field of tamper resistance is the dual problem of running trusted code on an untrusted host. Although of considerable practical value, there has been little formal work done on this problem. Most of the work reported in the literature is ad hoc. It is not clear that any solutions exist that have provable security guarantees. In addition, the field suffers from a lack of widely recognized standards to measure effectiveness. With these disclaimers in mind, we present a brief survey of some important work related to our self-checking technology.

Obfuscation attempts to thwart reverse engineering by making it hard to understand the behavior of a program through static or dynamic analysis. Obfuscation techniques tend to be ad hoc, based on ideas about human behavior or methods aimed to derail automated static or dynamic analysis. Collberg, et al. [CTL97,CTL98a,CTL98b] presented classes of transformations to a binary that attempt to confuse static analysis of the control flow graph of a program. Wang, et al. [Wan01,WDHK01,WHKD00] also proposed transformations to make it hard to determine the control flow graph of a program by obscuring the destination of branch targets and making the target of branches data-dependent. Wang, et al. present a proof that their transformations make the determination of the control graph of a transformed program NP-hard. Theoretical work on encrypted computation is also related to obfuscation. For example, Sander and Tschudin [ST98] propose a theoretical obfuscation method that allows code to execute in an encrypted form for a limited class of computations. Other work on obfuscation appears in [CGJZ01,NCJ01].

Customization takes one copy of a program and creates many very different versions. Distributing many different versions of a program stops widespread damage from a security break since published patches to break one version of an

executable might not apply to other customized versions. Aucsmith uses this type of technique in his IVP technology [Auc96]. Each instantiation of a protected program is different.

Software watermarking, which allows tracking of misused program copies, complements both obfuscation and customization by providing an additional deterrent to tampering. Many software watermarking methods have been proposed, but none of them appear to be in widespread use. Collberg and Thomborson [CT99] classify software watermarks as either static or dynamic and provide a survey of research and commercial methods. They make the distinction between software watermarking methods that can be read from an image of a program and those that can be read from a running program. Static methods include a system described by Davidson and Myhrvold [DM96] in which a program's basic blocks are shuffled to create a unique ordering in the binary, which serves as a unique identifier for that version of the program.

Self-checking, also referred to as tamper-proofing, integrity checking, and anti-tampering technology, is an essential element in an effective tamper-resistance strategy. Self-checking detects changes in the program and invokes an appropriate response if change is detected. This prevents both misuse and repetitive experiments for reverse engineering or other malicious attacks. Aucsmith [Auc96] presents a self-checking technology in which embedded code segments verify the integrity of a software program as the program is running. These embedded code segments (Integrity Verification Kernels, IVK's) check that a running program has not been altered, even by one bit. Aucsmith proposes a set of design criteria for his self-checking technology including interleaving important checking-related tasks for stealth (partial computation of the check sum), obfuscated testing code, non-deterministic behavior, customization of testing code (non-unique installations) and distributed secrets. We adhere to similar design criteria.

Chang and Atallah [CA] propose a method in which software is protected by a set of *guards*, each of which can do any computation. In addition to guards that compute checksums of code segments (analogous to our testers), they propose the use of guards that actually repair attacked code. Their emphasis is on a system for automatically placing guards and allowing the user of the system to specify both the guards and the regions of the program that should be guarded. In emphasizing these issues, their results are somewhat complementary to ours.

Collberg and Thomborson [CT00] provide a view of the nature of these classes of tamper-resistance technologies. Unfortunately, little research into the complementary aspects of these kinds of technologies can be found.

3 Design Objectives and Threat Model

The fundamental purpose of a dynamic program self-checking mechanism is to detect any modification to the program as it is running, and upon detection to trigger an appropriate response. We sought a self-checking mechanism that would be as robust as possible against various attacks while fulfilling various non-

security objectives. In this section we summarize our threat model and design objectives.

3.1 Functionality

- **Comprehensive and Timely Dynamic Detection** The mechanism should detect the change of a single bit in any non-modifiable part of the program, as the program is running and soon after the change occurs. This helps to prevent an attack in which the program is modified temporarily and then restored after deviant behavior occurs.
- **Separate, Flexible Response** Separating the response mechanism from the detection mechanism allows customization of the response depending upon the circumstances, and makes it more difficult to locate the entire mechanism having found any part.
- **Modular Components** The components of the mechanism are modular and can be independently replaced or modified, making future experimentation and enhancements easier, and making extensions to other executables and executable formats easier.
- **Platform Independence** Although the initial implementation of our self-checking technology is Intel x86-specific, the general mechanism can be adapted to any platform.
- **Insignificant Performance Degradation** The self-checking mechanism should not noticeably slow down the execution of the original code and should not add significantly to the size of the code. Our goal is to have no more than a 5% impact on performance.
- **Easy Integration** We designed our self-checking technology to work in conjunction with copy-specific static watermarking and with other tamper-resistance methods such as customization. Embedding the self-checking technology in a program relies on source-level program insertions as well as object code manipulations.
- **Suitable for a Large Code Base** Our test-bed executable was several megabytes in length.

3.2 Security

The two general attacks on a software self-checking mechanism are *discovery* and *disablement*. Methods of discovering such a mechanism, and our approaches for preventing or inhibiting these methods, follow.

Discovery

- **Static Inspection** We made the various components of the self-checking mechanism as stealthy and obfuscated as we could, to make detection by static inspection, especially automated inspection (by a program) hard.

- **Use of Debuggers and Similar Software Tools** Off-the-shelf dynamic analysis tools such as debuggers and profilers pose a significant threat to our self-checking technology. Self-checking requires memory references (reads) into executable code sections. These can be detected with a debugger, although any debugger that relies on modifying the code will be defeated by the self-checking mechanism. Our mechanism is enormously strengthened by the addition of a mechanism that detects standard debuggers and responds appropriately. (The design of such a mechanism is beyond the scope of this paper.) Homemade debuggers also pose a threat but require a substantial investment by an attacker.
- **Detection of Reads into the Code** We attempted to thwart both static and dynamic detection of reads into the code sections by obfuscating the read instructions, so that the code section addresses targeted by such reads were never in single registers. Detection of such reads thus requires noticing that such a read has actually occurred; inspecting the code or monitoring the registers will not reveal this fact.
- **Generalization** The self-checking mechanism consists of a large number of lightweight code fragments called *testers*, each testing a small contiguous section of code. An attacker, having discovered one such tester, could look for others by searching for similar code sequences. We customized the testers, so that generalizing from one to others is difficult: not only are there multiple classes of testers, each class performing a different test (computing a different hash function), but within each class the testers use different code sequences to do the same job.
- **Collusion** Our self-checking mechanism is designed so that it can be used on statically-watermarked code. If copy-specific watermarks are used, an attacker might be able to locate the tester mechanism by obtaining two differently marked copies of the code and comparing them. The differences might reveal not only the watermarks but also any changes needed in the self-checking mechanism to compensate for different watermarks. In our mechanism, the bits that vary in order to compensate for the watermarks are called *correctors*. These correctors are separated from the testers and the response mechanism. Therefore, neither the testers nor the response mechanism can be detected by collusion. In addition, detection of the correctors by collusion provides an attacker with very little information. Knowing the correctors and their values does not facilitate discovering or disabling the rest of the mechanism. The use of customization, in which there are many radically different copies of the code, would also foil this kind of attack since everything in the program looks different in every copy.
- **Inspection of Installation Patches** The final step of the watermarking and self-checking initialization process that we propose here relies on using a patch file to modify a previously obfuscated, non-functional executable. Inspection of the patch file might reveal some parts of the self-checking mechanism. With our method, the only parts of the self-checking mechanism that are in the patches are the correctors, not the testers or the response

mechanism. If copy-specific watermarking is not used, this patching process is not required.

Disablement In general, our goal was to eliminate single points of failure and to require discovery and modification of all or most of the self-checking mechanism for an attacker to succeed.

– **Modifying the Testers** One possible disabling attack is to modify one or more testers so that they fail to signal a modification of the tested code section. Our testers are designed to provide redundant, overlapping coverage, so that each tester is tested by several others. Disabling one or more of the testers by modifying them will produce detection of these changes by the unmodified testers. All or almost all of the testers must be disabled for this kind of attack to succeed.
– **Modifying the Response Mechanism** Another disabling attack is to modify the response mechanism. Again, because of the redundant testing mechanism, substantially all of the response functionality must be disabled for such an attack to succeed. In our current implementation we used direct calls to a tamper-response mechanism. Future possible work is to build a stealthier, more robust tamper-response mechanism, including variably delayed response and multiple paths to the response code.
– **Modifying Correctors** Another possible attack is to modify the code so that it behaves incorrectly and still does not trigger the testers. With our use of multiple overlapping hash computations, such an attack is unlikely to succeed without discovery of all or most of the testers. Such discovery would allow a successful tester-disabling attack. Thus, the former attack is no greater a threat than the latter.
– **Temporary Modifications** A dynamic attack might modify the code so that it behaves anomalously and then restore the code to its original form before the self-checking mechanism detected the change. Our use of dynamic, redundant self-checking minimizes this threat.

4 Algorithm Design

In this section we provide an overview of our self-checking mechanism, including some discussion of our design decisions and possible alternatives and extensions. In subsequent sections we discuss various aspects of the mechanism in more detail.

4.1 Components and Embedding Process

The self-checking mechanism consists of a collection of two kinds of components, *testers* and *correctors*, discussed in Sections 4.2. and 4.4, respectively. These components are embedded into an executable in a three-step process:

Step 1 Source-code processing Insert a set of testers, coded in assembly language, into the source code of the executable.

Step 2 Object-code processing

- **Step 2A** Shuffle groups of basic blocks of the object code, thereby randomizing the tester distribution.
- **Step 2B** Insert correctors, at least one per tester, into the object code.
- **Step 2C** Associate a corrector and a tester interval with each tester, in such a way as to provide redundant coverage of the executable and so that the correctors can later be set in an appropriate order to make the testers test correctly.

Step 3 Installation-time processing

- **Step 3A** Compute watermark values.
- **Step 3B** Compute corrector values given the watermark values.
- **Step 3C** Form patches containing the watermark and corrector values.
- **Step 3D** Install the program by combining the patches with a pre-existing, non-functional executable to prepare a watermarked, self-checking, fully functional executable.

Testers are inserted into source code instead of object code. If instead we were to insert the testers into the object code it would be difficult to insure that the registers used by the testers do not conflict with the registers being actively used by the object code at the insertion point. By inserting the testers in the source code, the compiler will do the appropriate register allocation to avoid any conflicts. This insertion method also affords us more control over the run-time performance of the self-checking mechanism, since we can more easily place testers in code segments with desired performance characteristics. On the other hand, we do not have fine-grained control over the placement of testers in the executable. Object-level placement of the correctors gives us great control over their static distribution, which is their most important attribute. The issues of where and when to insert the testers, correctors and other security components deserve further study.

Our self-checking method is designed to work in combination with watermarking. Since copy-specific watermarking must be done at installation time, the self-checking mechanism must either avoid checking the watermarks or must be modified at installation time to correct for the watermark values. We chose the latter course as being more secure. Our installation mechanism uses an "intelligent patching" process, in which both watermarks and correctors for the self-checking mechanism are placed into a set of patches on the server side. These patches are sent to the client, which patches the code to produce a working executable. The patches contain no information about the code outside the patches. This minimizes security risks on the client, time and space transferring the patch lists, and time and space on the server, for maintaining and computing patch

lists. This design led to a choice of linear hash functions for the self-checking mechanism. If copy-specific watermarking is not used, or an entire copy of the executable can be delivered at installation time, then the patching mechanism is not needed.

4.2 Testers

The heart of the self-checking mechanism is a collection of *testers*, each of which computes a hash (a pseudo-random many-one mapping) of a contiguous section of the code region and compares the computed hash value to the correct value. An incorrect value triggers the response mechanism.

To set the testing frequency and the size of the code tested by each tester, we need to balance performance, security, and stealth objectives. Experiments on a set of Pentium processors for a variety of linear hashes suggested that performance is relatively invariant until the size of the code interval being tested exceeds the size of the L2 cache. With our Pentium II processors we observed a marked deterioration of performance when the code interval size exceeded 512 kilobytes. Breaking the computation into pieces also addresses our threat model and meets our design objectives. It makes the self-checking mechanism stealthier. The testers execute quickly, without observable interruption to the program execution. Each of our testers, therefore, tests a contiguous section that is a few hundred kilobytes long.

A single tester, when executed, completely computes the hash value for its assigned interval and tests the result. We considered more distributed alternatives, in which a single call of a tester would only partially compute a hash value. Aucsmith promotes this type of design in his Integrity Verification Kernel, a self-checking mechanism proposed in [Auc96]. He promotes the use of *interleaved tasks* that perform only partial checking computations. With such an alternative, either a single tester or several different testers are responsible for the complete computation of the hash of an interval. We rejected such alternatives as being more complicated and less stealthy, in that they require storage of extra state information (the partially computed hash function).

An important design decision was where to store the correct hash values. One possibility is with the testers themselves. This poses a security risk. Because the self-checking mechanism tests the entire code and watermarks differ among different copies of the code, many of the hash values will differ among copies. In the absence of code customization (which creates drastically different versions of the code), the hash values can be exposed by a collusion attack, in which different copies of the code are compared. Storing the hash values with the testers thus potentially exposes the testers to a collusion attack. Another difficulty is the circularity that may arise if testers are testing regions that include testers and their hash values: there may be no consistent way to assign correct hash values, or such an assignment may exist, but be very difficult to compute.

Another possibility that avoids both of these problems (revealing the testers by collusion and circularity of hash value assignments) is to store the hash values in the data section. But then the hash values themselves are unprotected from

change, since the self-checking mechanism does not check the data section. We could avoid this problem by dividing the data section into fixed data and variable data, storing the hash values in the fixed data section, and testing the fixed data section, but this alternative may still be less secure than the one we have chosen.

We chose a third alternative, in which each hash interval has a variable word, called a *corrector*. A corrector can be set to an arbitrary value, and is set so that the interval hashes to a fixed value for the particular hash function used by the tester testing the interval. Collusion can reveal the correctors, but does not reveal the testers. Since the correctors are themselves tested, changing them is not an easy job for an attacker. Each tested interval has its own corrector, and is tested by exactly one tester. Aucsmith's testers (IVK's), although encrypted, are vulnerable to discovery by collusive attacks because the testers themselves are unique in each different copy of the protected software.

We experimented with multiple testers testing the same interval but rejected this approach as being overly complicated and not providing additional security.

Another important design decision is how to trigger the execution of the testers. We chose to let them be triggered by normal program execution, sprinkling them in-line in the existing code. Alternatives include having one or more separate tester threads, or triggering testers by function calls, exceptions, or some other specific events. We rejected the latter mechanisms as being insufficiently stealthy. Having separate tester threads in combination with an in-line triggering mechanism deserves further study, as it may provide additional security through diversity.

A third design decision was the choice of hash functions. We used chained linear hash functions: linearity was important to make installation easy. Because the actual hash values are not known until installation time, partial hash values had to be pre-computed and later combined with the values of the software watermarks. We chose to use multiple hash functions, so that knowing a hash interval and a corrector site is still not enough information to set a corrector value to compensate for a code change.

4.3 Testing Pattern

We cover the entire executable code section with overlapping intervals, each of which is tested by a single tester. The overlap factor (number of different testing intervals containing a particular byte) is six for most bytes. The testers are randomly assigned to the intervals. The high overlap plus the random assignment provide a high degree of security for the testing mechanism: changing even a single bit requires disabling a large fraction of the testers to avoid detection, even if some of the testers are ineffective because they are executed infrequently.

4.4 Correctors and Intervals

Each interval requires its own corrector, whose value can be set so that the interval hashes to zero. In our current implementation, each corrector is a single 32-bit unsigned integer. We place correctors in-between basic code blocks using

post-compilation binary manipulation. Everything between basic blocks is dead code; control will never be transferred to the correctors. An alternative would be to insert correctors as live code no-ops. We chose the former approach as being simpler and possibly stealthier, but this issue deserves further study.

Correctors are inserted as uniformly as possible throughout the code. Intervals are then constructed based on the desired degree of interval overlap, using randomization to select interval endpoints between appropriate correctors. This construction is such that it is possible to fill in corrector values in a left-to-right pass to make each of the intervals hash to zero. That is, there are no circular dependencies in the equations defining the corrector values. Since our hash functions are linear, an alternative approach is to allow such circularities and to solve the resulting (sparse) system of linear equations to compute corrector values. This alternative deserves further study.

Computing corrector values requires invertible hash functions, since we must work backwards from the desired hash value to the needed corrector value. This issue is discussed further below.

4.5 Tamper Response

A final component of our self-checking technology is the mechanism that invokes action if tampering is detected. In our current implementation each tester calls a tamper response mechanism directly via a simple function call.

We considered several alternative, indirect response mechanisms that appear to be promising. One of our primary objectives for the response mechanism is to avoid passing the execution of a response through a single point of failure. One of our primary integration objectives, however, was to make our mechanism easy to combine with other software protection mechanisms. Thus in our initial implementation we opted to use a simple direct response mechanism. Stealthier, more robust response mechanisms would use multiple access paths with a variable number of steps and running time. Such mechanisms are a subject of future work.

5 Tester Design and Placement

The function of a tester is to map a large interval of the executable (several hundred kilobytes) into a relatively small set of values (32 bits) by an easy-to-compute but "scrambling" map, a hash function. The idea is that any change to the interval being hashed will change the value returned by the tester and hence result in detection if the tester output is compared to the correct hash value. Of course, there is some probability of a changed interval mapping to the correct value, but with a good choice of hash function this chance will be vanishingly small (2^{-32}).

In this section we describe the design of our testers, including the objectives that went into the design. We also describe the approach we used to place the testers in the code.

5.1 Design Objectives

Our tester design was based on the following objectives.

- **Lightweight Hash Functions** Part of our design goal was that executing the tester mechanism should not have a substantial impact on program performance. This suggested the use of lightweight hash functions, combining an additive chaining mechanism with a simple 32-bit mapping that uses a few arithmetic or logical operations. In addition to being fast, simple hash functions were judged to be stealthier than standard hash functions such as CRC32 or SHA, since they consist of very short sequences of instructions.
- **Multiple Hash Functions** As a defense against an attack based on knowing the hash function, we thought it valuable to have multiple hash functions, randomly selected. A multiplicity of lightweight hash functions was judged to be sufficiently secure for our needs; cryptographically strong hash functions buy us little or nothing here.
- **Summarizable Hash Functions** Because watermark values and corrector values are computed in a final pass, at installation time, performance considerations dictated that partial hash functions be easily summarizable. This issue, which arose because of installation constraints, is discussed in more detail in Section 5.2.
- **Stealthy Testers** We wanted the testers to be hard to find by manual or automatic code inspection. In particular, they should look like any other part of the executable. Obfuscation techniques applied to the executable could conceivably help hide the testers. Short testers were judged to be stealthier than long testers.
- **Resistance to Auto-collusion** In an auto-collusion attack, the finding of one tester is used to help find others, based on their similarity to the first. To make an auto-collusion attack harder, we favored short customized testers. Ideally, each tester should be as different from the others as possible.
- **Obfuscated Address Calculation** A vulnerability of testers is that they load bytes from the code, which "normal" code does not do. We required that code-section addresses never appear in any general-purpose register during the calculation, thus complicating a "sampling" attack, in which the contents of the registers and the stack are monitored for suspicious values such as code-section addresses. Complex addressing modes provide one way of implementing this idea.
- **Harmless to Development** Testers are inserted into the source code and later (post-compilation) "stamped" with the start and end addresses of the intervals to be hashed. Unstamped testers should not trigger the response mechanism, so that unstamped testers are harmless to program development.

5.2 Linear Hash Functions

We did performance-testing with several lightweight hash functions built from one or more arithmetic or logical operations. We compared the performance of

each hash function with CRC32, a standard 32-bit chained hash function. Our sample hash functions ran 8-10 times faster than CRC32. We built "debug" testers using an "exclusive-or" chained hash function. The debug testers ran in 1-2 milliseconds per 128k bytes on a 200 Mhz Pentium. This is an upper bound on the expected performance of production testers, since the debug testers gathered extra information for use in our development. The debug testers were certainly fast enough that adding our self-checking mechanism to a program would not significantly impact its performance.

The requirements of invertibility and summarizability led us to the use of chained linear hash functions. In particular, given an interval of data d, consisting of the words $d_1, d_2, \ldots d_n$, the value $h_n(d)$ of the hash function on d is defined recursively by $h_0(d) = 0, h_i(d) = c * (d_i + h_{i-1}(d))$ for $0 < i \leq n$, where c is a suitably chosen non-zero multiplier that defines the hash function. Such a hash function is easily invertible, since we have $h_{(i-1)}(d) = h_i(d)/c - d_i$ for $0 < i \leq n$,which can be used recursively to compute $h_i(d)$ for any value of i, given $h_n(d)$.

Furthermore, the hash function is easily summarizable in the following sense. If we generalize the recurrence defining h to $h_0(x, d) = x, h_i(x, d) = c * (d_i + h_{i-1}(x, d))$, and view d as a constant vector and x as a variable, then $h_n(x, d)$ is a linear function of x. Namely, $h_n(x, d) = a_n(d)x + b_n(d)$, where a_n and b_n are defined recursively by $a_0(d) = 1, b_0(d) = 0, a_i(d) = c * a_{i-1}(d), b_i(d) = c * (d_i + b_{i-1}(d))$, for $0 < i \leq n$. Finally, the inverse function of h_n is also linear, and can be defined recursively in a similar way.

Invertibility and summarizability mean that, given an interval that is mostly constant but has certain variable words (watermark slots) and a single "corrector" word, we can precompute a representation of the hash function that requires space linear in the number of watermark slots. Given values for the watermark slots, we can then compute a value for the corrector that makes the entire interval hash to zero, in time proportional to the number of watermark slots. The precomputation time to construct the summary of the hash function is linear in the length of the interval. This computation is the final step in activating the testers. One problem in the actual corrector computation for Intel x86 executables is that the corrector is not necessarily aligned on a word boundary relative to the start and end of the hashed interval. This can, however, be handled, at a cost of complicating the calculation. Another possibility, which we did not choose, is to explicitly align the correctors, if necessary by providing 7-byte corrector slots rather than 4-byte slots.

The constant multipliers used to define our hash functions were chosen from a small set that allowed the hash computation to be performed without an explicit multiply instruction. Our particular construction resulted in a collection of 30 possible hash functions, corresponding to different multipliers. To expand the set of possible hash functions, we could have included an additive constant in the hash function (either by redefining the initial condition to be $h_0(d) = r$ or by redefining the recurrence to be $h_i(d) = c * (d_i + h_{i-1}(d) + r)$, for $0 < i \leq n$). This would increase the set of possible hash functions to $30 * 2^{32}$ and might be something to explore in the future. For now, having 30 different hash functions

was judged to be sufficiently secure, because an attacker must know not only the hash function but the start and end of the hashed interval, which seems as hard to determine as finding the tester itself.

5.3 Tester Construction and Customization

To help make our testers stealthy, we implemented a tester prototype in C and compiled it to get an assembly-language tester prototype. By doing this, we hoped to minimize the presence of unstealthy assembly-language constructs, specifically those that would not be generated by a compiler. However, in order to make the resulting testers practical, we made three modifications to this compiled tester prototype. First, we modified the prototype so that an unstamped tester would not call the response mechanism. Second, we added an obfuscation variable to the address calculations to guarantee that no code-section address would ever appear in a general-purpose register during the running of a tester (indicating a read of a code-section address). Third, we simplified the tester slightly.

Then, we applied a variety of customizations to guarantee that each tester had a unique code sequence, thus increasing the difficulty of an auto-collusion attack. These customizations included changing the multiplier defining the hash function and the exact instructions used to compute the hash function, shuffling the basic blocks of the tester, inverting the jump logic of conditional jumps, reordering instructions within basic blocks, permuting the registers used, and doing customization of individual instructions. The result was a set of 2,916,864 distinct tester implementations, each occupying less than 50 bytes.

5.4 Tester Placement

As discussed in Section 4.2, we chose to place testers in-line in the code and have them fire as they are reached during normal execution. Our goal for tester firing is that testers execute frequently enough that most or all of the code is tested often during normal execution, but not so often that tester firing causes a significant efficiency degradation. In order to place testers most effectively to realize these conflicting performance goals, we used source-level tester placement. Our tester placement strategy required significant manual effort. With more advanced software tools, the process could become more automated.

Our goal was to insert the various individual testers in source program functions so that the testers executed to meet coverage objectives in what we deemed to be typical program runs. To achieve this we used profiling tools to count function executions during runs of a several-megabyte test executable. We discarded functions not run at least once during start-up and at least once after start-up. We ordered the remaining functions in increasing order by execution frequency, and inserted testers into the functions in order, one tester per function, until the desired number of testers, around 200, were inserted.

This placement of testers, when combined with our method of interval construction and tester-to-interval connection, resulted in acceptable dynamic testing coverage, as we discuss in Section 6. A significant drawback, however, is that the testers are bunched in the executable, because they tend to be inserted into library functions that appear together in the executable. To overcome this problem, we relied on block-shuffling of the executable to disperse the testers more uniformly.

A straightforward analysis, which we omit here, shows that random shuffling of code blocks, assuming uniform block size and at most one tester per block, results in a maximum gap between testers that exceeds exactly equal spacing by a logarithmic factor. We deemed this adequate to provide the desired amount of testing robustness. (See Section 6.) We could achieve much more uniform spacing of testers by taking the location of the testers into account when doing the shuffling, or inserting the testers into the object code instead of the source code. This is a subject for future investigation.

6 Interval Construction

In addition to the testers, the other component of the self-checking mechanism is the code intervals over which the testers compute hash functions. Recall that we desire these intervals to provide uniform, redundant coverage of the entire executable and to be hard to discover. Also, each interval requires its own corrector, which must be able to be set so that the interval hashes to zero. Finally, there must be a global ordering of the correctors that allows them to be set sequentially, without circular dependencies.

We chose to base the interval construction on corrector placement. With this approach, interval construction consists of three steps: corrector placement, interval definition, and assignment of testers to intervals. We discuss these three steps in Sections 6.1–6.3. In Section 6.3 we also discuss the robustness of the resulting overlapping checking mechanism. In each section, we discuss alternatives to our current approach, both those we did and those we did not try.

6.1 Corrector Placement

We need one interval, and hence one corrector, per tester. Since we want the intervals to be approximately of equal size and approximately uniformly spaced, we want the correctors to be approximately uniformly spaced as well. Our current method of corrector placement is a second-generation design that inserts correctors as dead code (between basic blocks) once basic block shuffling is completed.

It is illuminating to consider our original design, which used source-code insertion, to understand the virtues of the current method. In our original design, and our original implementation, a very large number of correctors, consisting of obfuscated NOPs, were inserted into the source code by including them in appropriately chosen source-language functions. In the absence of basic-block

shuffling, the distribution of these correctors is extremely non-uniform; indeed, the correctors are often clumped closely together. We therefore relied on shuffling of basic blocks to provide a much more uniform distribution of correctors in the executable. Even random shuffling does not produce uniformly spaced correctors; the corrector gaps have a Poisson distribution, which implies that the expected maximum gap size is a logarithmic factor greater than the average gap size. To overcome this problem we inserted many more correctors than needed (at least a logarithmic factor more) and used a "pruning" step to select a small subset of correctors that we actually used.

Although we implemented this method and demonstrated its effectiveness in practice, it has at least three drawbacks: the insertion of many more correctors than needed, the extra computation step of corrector pruning, and the need to carefully place correctors in replicated functions in the source code to make sure there are enough correctors in the executable. It was the last drawback that made us replace this corrector insertion method with the one described below. In the course of our experiments, we discovered that the correctors were being inserted into code that was never executed. Eliminating this dead code significantly shrank the size of our test executable but left us with no convenient place in the source code to put the correctors.

We therefore replaced our corrector insertion method with an executable-based insertion method. Specifically, the correctors are inserted after the blocks are shuffled. This approach has some significant advantages over the source-code-based insertion scheme. It gives us fine-grained control over the positioning of the correctors in the executable. We can insert correctors as dead code (between basic blocks) instead of, or in addition to, as obfuscated NOPs. Dead-code correctors can consist just of the 32 correction bits, rather than forming valid instructions or instruction sequences. We also can dispense with the corrector pruning step (although we left this step in our current implementation: it provides possibly redundant smoothing of the corrector distribution).

In detail, the corrector placement process works as follows. Word-length (32 bit) corrector slots are inserted at the end of basic blocks (after unconditional jump instructions). We chose a gross number of corrector slots to insert (before pruning). To determine where to insert the correctors, we count the total number of usable basic blocks for insertion and divided by the number of correctors. If the result is k, we insert a corrector after each k basic blocks.

We then prune the correctors down to the set actually used, as follows. While we have too many correctors, we apply the following step to remove a corrector: find the two adjacent correctors closest together (in bytes) and eliminate the one whose removal creates the smallest new gap. This algorithm can be implemented efficiently using a heap (priority queue) data structure to keep track of the gap sizes, at a logarithmic cost per deleted corrector. In our current performance run, we use 1000 as the gross number of correctors and about 200 as the number of testers and net number of correctors.

An improved design, which we leave for future work, is to space the correctors using a byte count (instead of a block count) and to eliminate the pruning step

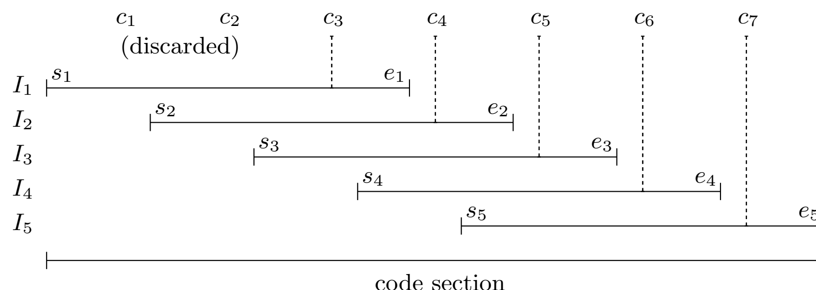

Fig. 1. Illustration of interval construction for n=5 and k=3

entirely. We also would like to investigate whether insertion of testers inside basic blocks, rather than just between basic blocks, provides sufficient additional uniformity as to be worthwhile. (Since basic blocks are extremely small compared to interval lengths, we doubt it.) An interesting research question is to devise an efficient algorithm to insert k correctors among n basic blocks so as to minimize the maximum gap (in bytes) between correctors (assuming dummy correctors exist at the start and end of the code).

6.2 Interval Definition

We define the intervals to be tested based on the placement of the correctors, using random choice of interval endpoints between appropriate correctors to help make it hard for an attacker to determine these endpoints. In addition we use a *overlap factor* $k \geq 1$, such that most bytes in the executable will be covered by k intervals. Currently, we use an overlap factor of 6.

Suppose we desire n test intervals $I_i, 1 \leq i \leq n$. Then we will use $n + k - 1$ correctors, of which $k - 1$ will be discarded. Label the correctors $c_1, c_2, ..., c_{n+k-1}$ in the order they occur in the executable. We choose a start s_i and end e_i for each of the intervals, as follows. Start s_1 is at the beginning of the code section, and end e_n is at the end of the code section. For i in the range $1 < i \leq k$, we choose s_i uniformly at random between c_{i-1} and c_i and e_{n-i+2} uniformly at random between c_{n+k-i} and $c_{n+k-i+1}$. For i in the range $k < i \leq n$, we choose two points uniformly at random between $c_{i-1}$and c_i. The smaller point is s_i and the larger point is e_{i-k}. We then associate corrector c_{k+i} with interval I_i, and discard the first $k - 1$ correctors. The construction is illustrated in Figure 1.

This choice of intervals has two important properties. Except near the beginning and end of the code section, every byte of the code is contained in k (or possibly $k+1$) test intervals. The first corrector in test interval I_i is c_{i+k-1}, which means that we can set corrector values in the order $c_k, c_{k+1}, \ldots c_{n+k-1}$ to make

successive intervals $I_1, I_2, \ldots, I_n$ hash correctly without having later corrector settings invalidate earlier settings. That is, there are no circular dependencies.

The ends of the code section are not covered as redundantly as the rest of the code. We do not think this affects the robustness of the testing mechanism in any significant way. It is easy to modify the interval construction scheme so that the corrector setting works from the two ends of the code toward the middle, so that the non-redundant coverage occurs in the middle. We could choose the region of non-redundant coverage randomly, or to be unimportant code. Also, as noted in Section 4.4, we could modify the interval construction to allow circularities and solve the resulting system of linear equations to get corrector values. Such a method would be even more robust. Modifying our scheme along these lines is a topic for future research.

6.3 Assignment of Testers to Intervals

Once intervals are constructed, the next step is to assign testers to the intervals. The objectives of this assignment are coverage and security. We want each byte to be tested often as the code is running, and we want to force an attacker to disable many or most testers to successfully modify even a single byte of the program without detection. Our approach to accomplishing these goals is to harness the power of randomization: we assign each tester to a different interval using a random permutation to define the assignment.

Both experimental and theoretical evidence suggest that a random assignment is a good one. Our theoretical evidence is based on our intuition and on theorems about random graphs in the literature. An interesting area for future research is to develop a mathematical theory, with proofs, about the performance of random tester assignment. For now, we rely on the following observations.

First, almost every byte in the code is covered by k testing intervals and hence tested by k testers. With random assignment, the most important bytes will be redundantly tested, even if a significant fraction of the testers are ineffective because of infrequent execution.

Our second and third observations concern a graph, *the tester graph*, that models the pattern of testers testing other testers. The vertices of the graph are testers. The graph contains an edge from tester A to tester B if tester B is contained in the interval tested by tester A. (Our construction of intervals allows the possibility that an interval boundary might be in the middle of a tester. In such a case the graph would not contain the corresponding edge. We could easily modify our interval construction to move interval boundaries outside of testers. Whether this is worth doing is a subject for future research.)

Suppose that all testers are effective (i.e., they execute frequently when the program is running normally). Suppose further that an attacker modifies a byte of the program that is in an interval tested by tester X. Then, to avoid detection, the attacker must disable every tester Y such that there is a path from Y to X in the tester graph. Suppose the tester graph is *strongly connected*; that is, there is a path from every vertex to every other vertex. Then a successful attack, changing even a single byte, would require disabling *every* tester.

We thus would like the tester graph to be strongly connected. With our method of interval construction and random tester assignment, the tester graph is strongly connected with high probability. This is true as long as the intervals are sufficiently uniform and the redundancy factor k is sufficiently high. Experiments confirmed that the number of components drops rapidly as k increases. For small values of k, there is one large component and a small number of single node components. (Thus it is close to strongly connected.)

If strong connectivity were the only desired property of the tester graph, random assignment would not be necessary. We could, for example, guarantee strong connectivity by embedding a large cycle in the tester graph. Strong connectivity is not enough to guarantee the robustness of the testing mechanism, however. For example, if the tester graph consists only of one big cycle and some testers are ineffective (meaning they are in parts of code that do not get executed during an attack), then the effective tester graph consists of disconnected pieces, and (hypothetically) certain parts of the program may be attacked by disabling only a few testers.

A stronger connectivity property is that, even if a fraction of the testers are ineffective, a single byte change would require disabling many or most of the effective testers to avoid detection. This kind of robust connectivity is related to the expansion property, which is possessed by certain random graphs. "Expansion" means that there is a constant factor $\alpha > 1$, such that for any subset X of at most a constant fraction of the vertices, at least $\alpha|X|$ other vertices have edges into X. Expansion implies both strong and robust connectivity, depending on α. The possession of this property by random graphs is the main reason we used random tester assignment. The expansion property is hard to test empirically (doing so takes exponential time), so we have not verified that our tester graphs possess it. Nor have we done a careful theoretical analysis (which would yield very conservative constant factors). This is future research. We are confident, however, that our tester graphs possess sufficiently robust connectivity, and we hope to investigate this issue further.

7 Summary and Future Work

We have designed and built a dynamic software self-checking mechanism suitable to protect client-side software running in a potentially hostile environment. It is designed to be used in conjunction with other tamper-resistance techniques, and integrated with static copy-specific software watermarking.

Directions for future research include building a stealthier response mechanism that would add an additional layer between response detection and response reporting, doing further experimental and theoretical work on the coverage and robustness of the self-checking mechanism, modifying and simplifying the corrector insertion step, and developing additional hash functions, customizations, and obfuscations for the testers. A more speculative but potentially interesting direction is to investigate non-hash-based checking, in which, for example, the testing mechanism checks that correct values are stored in certain registers at

certain times. Other directions include exploring other triggering mechanisms for the testers, e.g., executing some of them as separate threads, investigating temporally-distributed testers, and studying hash-based methods that do not use correctors.

References

[Auc96] D. Aucsmith. Tamper resistant software: An implementation. In R.J. Anderson, editor, *Information Hiding, Lecture Notes in Computer Science 1174*, pages 317–333. Springer-Verlag, 1996.

[CA] H. Chang and M. Atallah. Protecting software by guards. This volume.

[CGJZ01] S.T. Chow, Y. Gu, H.J. Johnson, and V.A. Zakharov. An approach to the obfuscation of control–flow of sequential computer programs. In G.I. Davida and Y. Frankel, editors, *ISC 2001, Lecture Notes in Computer Science 2200*, pages 144–155. Springer-Verlag, 2001.

[CT99] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages, San Antonio, TX*, pages 311–324, January 1999.

[CT00] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, obfuscation – Tools for software protection. Technical Report 2000–03, University of Arizona, February 2000.

[CTL97] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, 1997.

[CTL98a] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages, Chicago, IL*, pages 28–38, May 1998.

[CTL98b] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *Principles of Programming Languages 1998, San Diego, CA*, pages 184–196, January 1998.

[DM96] R. Davidson and N. Myhrvold. Method and systems for generating and auditing a signature for a computer program, September 1996. US Patent 5,559,884. Assignee: Microsoft Corporation.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[NCJ01] J.R. Nickerson, S.T. Chow, and H.J. Johnson. Tamper resistant software: extending trust into a hostile environment. In *Multimedia and Security Workshop at ACM Multimedia 2001, Ottawa, CA*, October 2001.

[Nec98] G.C. Necula. *Compiling with proofs*. PhD thesis, Carnegie Mellon University, September 1998.

[NL96] G.C. Necula and P. Lee. Safe kernel extensions without run–time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, WA*, pages 229–243, October 1996.

[ST98] T. Sander and C. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security, Lecture Notes in Computer Science 1419*. Springer–Verlag, 1998.

[Wan01] C. Wang. *A security architecture of survivable systems*. PhD thesis, Department of Computer Science, University of Virginia, 2001.

[WDHK01] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software–based survivability mechanisms. In *IEEE/IFIP International Conference on Dependable Systems and Networks, Goteborg, Sweden*, July 2001.

[WHKD00] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing the static analysis of programs. Technical Report CS–2000–12, Department of Computer Science, University of Virginia, 2000.