Detecting Patching of Executables without System Calls

Sebastian Banescu
Mohsen Ahmadvand
Alexander Pretschner
Technische Universität München, Germany
{banescu,ahmadvan,pretschn}@cs.tum.edu

Robert Shield
Chris Hamilton
Google Inc.
{robertshield,chrisha}@google.com

ABSTRACT

Popular software applications (e.g. web browsers) are targeted by malicious organizations which develop potentially unwanted programs (PUPs). If such a PUP executes on benign user devices, it is able to manipulate the process memory of popular applications, their locally stored resources or their environment in a profitable way for the attacker and in detriment to benign end-users. We describe the implementation of a tamper detection mechanism based on code self-checksumming, able to detect static and dynamic patching of executables, performed by PUPs or other attackers. As opposed to other works based on code self-checksumming, our approach can also checksum instructions which contain absolute addresses affected by relocation, without using calls to external libraries. We implemented this solution for the x86 ISA and evaluated the performance impact and effectiveness. The results indicate that the run-time overhead of self-checksumming grows proportionally with the level of protection, which can be specified as input to our implementation. We have applied our implementation on the Chromium web-browser and observed that the overhead is practically unobservable for the end-user.

CCS Concepts

•**Security and privacy** → **Software security engineering;**

Keywords

Software protection; Tamper detection; PUPs

1. INTRODUCTION

Code patching is performed for various reasons and by various stakeholders of a software application. Incremental updates are a typical example. Similarly, attackers patch the code of a software application either statically or during runtime, in order to change the program's behavior. Historically, this kind of attack was mainly aimed at cracking license checks in computer games, which was detrimental for the profit of the game vendors. Starting from the late 2000s some organizations started to automate such code patching attacks targeting popular applications (e.g. web browsers) in order to change their behavior in a way that would bring financial gains to those organizations.

Such automated attacks fall into a category called *potentially unwanted programs* (PUPs). PUPs are often bundled together with (seemingly) useful software, which leads end-users into unknowingly installing them. Once installed, PUPs change a program's behavior by tampering with process memory, locally stored resources or the environment in which they run. For instance, they change the default search engine of a web-browser, aggressively display pop-up advertisements, track actions of end-users, cause an overall system slowdown and ask for fees to "fix performance."

Recent work on PUPs indicates that Google Safe Browsing generates over 60 million warnings related to PUPs per week, three times that of malware warnings [33]. Techniques employed by PUPs (e.g. code injection in the process memory, run-time memory patching, system call interposition) generally, do not raise any alarms in anti-virus software because they are also performed by non-malicious third party software including anti-virus software, accessibility and graphics driver tools [32]. Some anti-virus products are able to detect PUPs. However, the vendors of popular software applications (e.g. web browsers) cannot assume that such anti-virus software is present on all end-user systems. Therefore, developers of popular applications aim to incorporate light-weight software protection mechanisms inside of their own products, i.e. mechanisms that introduce a tolerable amount of overhead and are transparent for end-users.

This paper presents a mechanism that detects code patching attacks at runtime. The idea is based on software self-checking [5, 16] which can detect code tampering attacks, without communicating with a trusted server. The idea is to create a white-list containing checksums of pieces of code, which are invariant from one execution to another on any fixed OS version and at various states during execution. Integrity checks are interleaved with existing code and verify if invariants hold during execution. If these invariants do not hold, then process memory modification has been detected and a response action is executed. To prevent patching attacks on the code that performs the checksums itself, our integrity checks form a strongly connected network where multiple checks protect other checks.

We make the following contributions:

- Extending the state of the art on self-checking, we propose a way of checksumming instructions that contain

CODASPY'17, March 22 - 24, 2017, Scottsdale, AZ, USA

ISBN 978-1-4503-4523-1/17/03…$15.00

DOI: http://dx.doi.org/10.1145/3029806.3029835

absolute addresses, which may change each time the program is loaded into memory by the OS loader. Moreover, our idea does not employ system calls, which are susceptible to system call interposition attacks [12].

- We provide an open source implementation on top of the *Google Syzygy Transformation Toolchain* [14], which can be applied directly to binary executables.[1]
- We evaluate the effectiveness against attackers who are aware of the details of our protection mechanism and show that some attacks do not scale while others can be countered by obfuscating the code of checks.
- We evaluate the performance overhead that our mechanism has on multiple types of applications, including Chromium. We show that the overhead is acceptably small for applications that are not CPU intensive.

The rest of this paper is organized as follows. §2 presents related work. §3 describes the general design of our approach, while §4 presents its implementation for x86 Assembly language. §5 presents the evaluation of our implementation, and §6 concludes and gives directions for future work.

2. RELATED WORK

Software tamper protection consists of mechanisms that detect or prevent unauthorized modifications of software. One simple form of tamper protection is binary whitelisting [23] which checks the hash of a binary against a securely managed list. A second technique uses the currently executing code as a decryption key for code that is executed next [25]. This mechanism can be circumvented because the attacker can get all the correct keys and code by monitoring the executions of a non-patched program. Other tamper protection techniques are based on software self-checking [5, 16, 13], which have been successfully combined with self-modifying code. Unfortunately, self-modifying code as a defense mechanism requires memory pages which are both writable and executable, which enables remote code injection attacks [37].

Self-checksumming [5, 16] protects against tampering by adding code to an application. This code reads other parts of the code and compares their checksums to precomputed values. Junod et al. [21] have implemented a tamper protection mechanism based on the same self-checksumming techniques as our work. They do not include any details on how absolute addresses in code are handled, which is one of the main contributions of this paper.

Oblivious Hashing (OH) [6], computes a checksum over the dynamic state of an execution trace (e.g. code counters, memory values, branch conditions, etc.). This offers higher stealth than self-checking, because OH does not imply unusual execution patterns like self-checksumming does, i.e. a program reading its own code. However, OH has some shortcomings that makes it unusable for many applications, e.g. OH cannot handle branches based on program inputs.

Tamper protection via communication with trusted servers is employed in massive multiplayer online games (MMOGs) to detect cheating. Anti-cheat software such as PunkBuster [10], Valve Anti-Cheat (VAC) [35], Fides [22] and Warden [15] perform client-side computation, which are validated by a trusted server. Pioneer [30] and Conqueror [24] work similarly as anti-cheat software but target the protection of legacy systems. Jakobsson and Johansson [19] propose a similar technique for detecting malware on mobile devices. Collberg et al. [8] propose tamper protection by pushing continuous updates from a trusted server to the client, which force the attacker to repeat reverse engineering and patching on each update. One disadvantage of these tamper protection techniques is their dependence on external trusted servers. This dependence may cause a denial-of-service to end-users of the protected software applications which are also meant to be used offline, in case Internet connectivity is unavailable. Our solution proposed operates locally, i.e., without dependence on a trusted server.

Tamper protection via *trusted computing* is usually enabled by trusted hardware. Intel has released a hardware based tamper resistance mechanism [1], known as *Software Guard eXtension* (SGX), which enables trusted computing. Morgan et al. [26] propose building a hypervisor to perform integrity checks at higher privilege levels than the attacker. Dewan et al. [9] also use a trusted hypervisor to protected the sensitive memory of programs. Feng et al. [11] propose performing randomly-timed stealthy measurements using Intel's Active Management Technology [17], which can be validated locally. These approaches provide high security guarantees. However, they require trusted hardware to be available and the installation of a hypervisor. Software developers of popular software (e.g. web browsers), generally do not want to restrict their user base by imposing such requirements.

Banescu et al. [2] as well as Blietz and Tyagi [4] propose tamper detection techniques based on runtime monitoring. The target program is transformed at compile time to report its control flow to a separate monitoring process, which verifies it according to a whitelist. The monitor can be protected using code hardening techniques because it is compact, e.g., by white-box cryptography [36] or control-flow obfuscation [29], without causing significant runtime overhead on the target program. However, this approach fails to detect code patches not violating control flow integrity, e.g. inline patching of sequential code. Moreover, this approach employs system calls, which are vulnerable to system call interposition attacks [12]. Our approach does not employ system calls and can detect inline patching of sequential code.

3. DESIGN

The idea behind software self-checking is to interleave *checkers* with the original code of an application. A checker is a piece of code which, firstly, reads a number of machine code bytes from different parts of the memory of the same process it is executing in. Checkers read continuous sequences of code bytes, so-called *blocks* of code, that we refer to as *checkees*. Secondly, the checker computes a *checksum* of those checkees, which we also refer to as *hash*. Thirdly, it compares the hash against a hard-coded *precomputed value* of the checkees. If the checksum matches the precomputed value, then normal execution continues, i.e., as in the original code. Otherwise, a *response function* is invoked. Typical response functions include halting execution (immediately or after a certain amount of time), degradation of output(s), logging the attack and/or restoring the patched code [20].

3.1 Checksumming Absolute Addresses

One challenge is checksumming absolute addresses which change dynamically each time the program is loaded in memory. This is illustrated in Figure 1, which shows x86 assembly code snippets (left column), their corresponding static ma-

[1] https://github.com/google/syzygy/tree/integrity

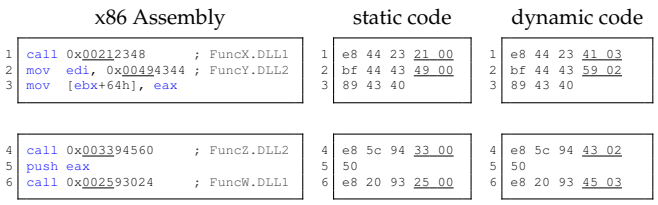


Figure 1: x86 Assembly vs. static and dynamic machine code.

chine code (middle column) and the machine code once it has been loaded in memory (right column). Note that the code does not fulfill any useful function and that similar problems occur for other CPU architectures, such as x64, ARM, MIPS, etc. The top snippet in Figure 1, starts with a call to the *FuncX* function from a dynamically loaded, shared library called *DLL1*. For ease of readability, we provide the function name in a comment following the absolute address of the function in the *call* instruction. The code snippet continues with moving the absolute address of the *FuncY* function from *DLL2* into the *edi* register, and one more *mov* instruction without any absolute references.

Statically, absolute addresses are constant, because they are the sum of a base address of a binary (which is zero before the program is loaded) and a constant offset (in that binary, e.g., a DLL), of the function being referenced. This means that the *precomputed* checksum value on the static code is fixed. However, at runtime absolute addresses change because executables are loaded at random base addresses, due to *address space layout randomization* (ASLR) [31], a software protection mechanism used against code injection attacks. ASLR loads binary executables and shared libraries at different memory locations if their preferred memory location is already occupied. For instance, Figure 1 shows that the underlined part of the absolute addresses (little-endian format) in the static and dynamic machine code are different. This is because after loading the code in memory, the base address of *DLL1* and *DLL2* are randomly assigned to 0x03200000, respectively 0x02100000. These base addresses get added to the offsets in the static code. Therefore, the checksum of the dynamic code from Figure 1, may be different every time the program is loaded in memory. The checksum computed at runtime then differs from the precomputed checksum, which (incorrectly) triggers the response function. This causes end-user annoyance and leads the software vendor to erroneously believe that code tampering has taken place.

3.2 Computing Invariant Checksums

A first approach to checksumming absolute addresses is to simply ignore all those machine code bytes which represent absolute addresses. This guarantees that any precomputed checksum will always be the same as the dynamically computed checksum. However, in order to compute such checksums at runtime, the checkers require information regarding the offsets of all bytes which represent absolute addresses. This increases the size of the checkers and lowers performance because these offsets need to be added to the code and used during checking. Moreover, it allows attackers to modify the ignored absolute addresses because they are skipped by self-checksumming.

We propose a different approach to checksumming absolute addresses based on the following two observations. Firstly, an *absolute address* a is the sum of the *base address* b of that PE loaded in memory and a *relative offset* o inside that PE, i.e., $a = b + o$. b may change whenever the PE is started. In contrast, o is constant and, as we have seen the same in the static and the dynamic cases.

Secondly, we can eliminate variable base addresses if we subtract two absolute addresses with the same base. Let a' be another absolute address with the same base address as a but a different offset o', i.e. $a' = b + o'$. Then we eliminate the common base address by subtracting two absolute addresses: $a - a' = (b+o) - (b+o') = o - o'$, and $o - o'$ is invariant across multiple runs of the same program. If two checkees contain absolute addresses a and a', respectively, then the difference between the two checkees will be constant. We are hence interested in finding a byte array checksumming function H which maintains this invariant, $a - a' = o - o'$:

$$H(a) - H(a') = H(o) - H(o'). \quad (1)$$

The left-hand side of Eq. 1 is computed while the protected program is running. The right-hand side can be precomputed from the static binary code, because the offsets in a PE do not change across different times that PE is loaded into memory by the OS. Therefore, we can hard-code $H(o) - H(o')$ as a precomputed checksum and use it at runtime.

Another solution for eliminating a variable base address from an absolute address is to somehow obtain the base address b of the dynamic libraries and subtract their values from the corresponding absolute address a, i.e. $H(a) - H(b) = H(o)$, which is constant. Obtaining base addresses can be done using system calls. However, attackers such as PUPs can easily detect and intercept ("hook") system calls via a technique called *system call interposition* (SCI) [12]. Using SCI a PUP can block or modify a system call such that it returns a different (incorrect) value.

Instead of using system calls, we propose statically inserting sequences of inconsequential instructions that reference the base address b of the binary or library, which is referenced by the absolute address $a = b + o$, whose base address needs to be canceled. Since the location of these inserted instructions is known, we can dynamically compute a hash of them, and know that they only contain one absolute reference, namely the base address b of the binary or library referenced by a. Hence, we can use this hash to eliminate the base address of a, i.e. $H(a) - H(b)$. However, this requires inserting additional code in the binary that we want to protect. Therefore, in order to reduce the amount of inserted instructions, we will only resort to inserting such instructions only when we cannot find a checkee containing an absolute address a' with the same base address as a, such that Eq. 1 cannot be applied.

3.3 Generalization

We can generalize this idea to the level of multiple instructions with references to multiple external dynamic libraries. To do this, the set of checkees associated with one checker is selected such that combining their checksums will cancel out all base addresses in absolute references according to the observation from Eq. 1. For example, the snippet of code (checkee) from the bottom of Figure 1 contains two instructions (lines 4 and 6) with absolute references to functions from the same DLLs[2] as the snippet (checkee) from the top of Figure 1.

[2]Technically speaking the absolute addresses from both snippets in Figure 1 are pointing to the so called *import address table* (IAT) in the data segment of the PE. The entries in the IAT actually contain absolute addresses to the ac-

This means that at runtime, the two *call* instructions from Figure 1, bottom, have the same base addresses as the first two instructions from Figure 1, top. If we denote the sequence of all dynamic code bytes of the checkees from Figure 1 as B_1 (top) and B_2 (bottom), then the value $H(B_1) - H(B_2)$ will be invariant across all application restarts.

This observation carries over to multiple blocks that call functions in DLLs with different base addresses multiple times. For instance, assume a PE with references to three DLLs having base addresses b_1, b_2, and b_3. Assume a (loaded) block B_3 referencing absolute addresses $b_1 + o_1$, $b_2 + o_2$, and $b_3 + o_3$; block B_4 referencing absolute addresses $b_2 + o_2$ and $b_2 + o_4$; and block B_5 referencing $b_1 + o_5$, $b_1 + o_6$, $b_3 + o_3$ and $b_3 + o_7$, for arbitrary constant offsets $o_1, \ldots, o_7$. Then we need to generate the linear combination $-2 * H(B_3) + H(B_4) + H(B_5)$, which is constant across multiple executions of the PE, because the base addresses cancel each other out.

More generally, let $\mathcal{B}$ be a set of blocks with $|\mathcal{B}| = n$, and a set of dynamically linked libraries (DLLs) $\mathcal{D}$ with $|\mathcal{D}| = d$. We can statically analyze how many times a block references functions from a DLL and store this information in matrix $Q \in \mathbb{N}^{n \times d}$. $Q(i, j)$ is the number of calls from block i to functions in DLL j. We will use the notation $Q(i)$ to denote the i-th row of Q. Let $B \in \mathbb{N}^d$ be the set of base addresses of the DLLs that are dynamically assigned at load time. $B(i)$ hence is the base address of DLL i. Remember that B cannot easily be obtained at runtime without using system calls (see §3.2). We now show how to exploit the idea of cancelling out base addresses with linear combinations.

3.3.1 Equation systems

Let $H_p \in \mathbb{Z}^n$ be the statically and $H_r \in \mathbb{Z}^n$ the dynamically computed checksums of each block, e.g., using function H that performs word addition modulo 2^N, where a word has N-bits. Because all base addresses are zero in the static code, we want to ensure $H_p(i) \equiv H_r(i) - \sum_{j=1}^{d} Q(i, j) * H(B(j)) \mod 2^N$ at runtime, for all $1 \leq i \leq n$, which we write as

$$H_p \equiv H_r - Q \cdot H(B) \mod 2^N \tag{2}$$

when using matrix multiplication notation. The idea now is to compute linear combinations of checksums that add up to zero. Let $x \in \mathbb{Q}^n$ be the respective vector of coefficients. We want to find values for the components of $x \neq \mathbf{0}$ such that $\forall 1 \leq j \leq d : \sum_{i=1}^{n} Q(i, j) * x(i) = 0$, which we rewrite as

$$Q^T \cdot x = \mathbf{0}, \text{ or } x^T \cdot Q = \mathbf{0} \tag{3}$$

where $\cdot^T$ denotes the transpose of a matrix. Multiplication of Eq. 2 with x^T then cancels out the base addresses:

$$x^T \cdot H_p \equiv x^T \cdot H_r - x^T \cdot Q \cdot H(B) \equiv x^T \cdot H_r \mod 2^N.$$

To perform integrity checks at runtime, we hence need to store only the non-zero values of the vector x satisfying Eq. 3 and the scalar $x^T \cdot H_p$.

Eq. 3 is a linear equation system for which it is easy to compute rational solutions (if they exist). However, for large numbers of blocks and rational coefficients, rounding errors are likely to materialize, and it is hard to predict their effects. It is therefore desirable to consider Eq. 3 as a linear Diophantine equation system and stipulate $x \in \mathbb{Z}^n$.

tual function entry points in dynamic libraries loaded at different locations in memory. This level of indirection is similar for other executable formats.

Linear Diophantine equation systems can effectively be solved using SMT solvers [3]. However, in our special case, effectiveness, scalability and efficiency quickly become concerns. Statically, the equation system may become too large to be handled by the SMT solver. The equation system need not have a non-zero solution, possibly not even a rational one. This can trivially be the case, for instance, if a DLL is called by exactly one block. Dynamically, if a frequently executed block needs to compute checksums of very many blocks, runtime performance becomes a concern.

3.3.2 Reducing large equation systems

Matrix Q is usually very sparse in practice. A first idea is thus to reduce Q to the dimension of its rank by computing the basis. However, this essentially is done by the SMT solver anyway, so we cannot expect too much effect here.

Because of the sparseness of Q, we can hope to compute a set of rather small submatrices for which we can solve the corresponding equation systems independently and in parallel. To do so, we choose subsets P of the rows of Q such that it is ensured that for each column of P, there are none or more than one non-zero elements. The interpretation is that we choose sets of blocks such that any DLL (column) is called either never (all elements zero) or by at least two blocks (at least two elements of this column are non-zero). By removing the all-zero columns we get a smaller equation system, and making sure that the sets P together cover all rows of Q, we have simplified the problem.

The algorithm takes as input a set of rows R of matrix Q and outputs a set of sets of rows $\{W_1, \ldots, W_k\} \subseteq 2^R$ such that (1) all rows of R are covered ($\bigcup_{i=1}^{k} W_i = R$) and (2) and for each set of rows (blocks) W_i and each column (DLL) d, d is either not called by any row in W_i, or there are rows $r_1 \neq r_2$ in W_i that both call d at least once. Each W_i is one independent equation system, usually much smaller than R.

The algorithm starts by initializing the counter of blocks of blocks (rows), $i = 0$ and consists of the following steps:

Step 0: Remove from R all rows that contain zeros only, all columns that contain zeros only, and all columns that contain exactly one non-zero element. The latter will be catered to by adding bogus calls to the respective DLL in a block that does not call any other DLL in §3.3.3.

Step 1: If $R = \emptyset$, stop. Otherwise, $W_i := \{r\}$ for a random row $r \in R$. W_i is the currently computed set of rows.

Step 2: Otherwise, if $R \neq \emptyset$, compute the *fitness* $\varphi(r', W_i)$ for each $r' \in R$ as described below. The row with the best fitness will be added to the current submatrix W_i. A fitness value of $-\infty$ indicates that the respective row does not help complete the equation system represented by W_i because there is no overlap with the DLLs called in W_i. Choose $r'' = \arg\max_{r' \in R}(\varphi(r', W_i))$ to be the row with the highest fitness.

Step 3: If $\varphi(r'', W_i) \neq -\infty$, let $W_i := W_i \cup \{r''\}$ and $R := R \setminus \{r''\}$ (because further blocks of the partition will usually not need to consider r''). If W_i now is such that every DLL is either called by no or by at least two rows, the current block of rows is ready. Let $i := i + 1$ and goto step 1. Otherwise goto step 2.

Step 4: Otherwise, if $\varphi(r'', W_i) = -\infty$, we need to resort to a row that has been picked before. Compute $\varphi(r', W_i)$ for all rows $r' \in \bigcup_{j<i} W_i$ that have already been picked earlier, and pick the best: $r''' = \arg\max_{r' \in \bigcup_{j<i} W_j}(\varphi(r', W_i))$.

Step 5: Because we have gotten rid of rows that make

equation systems inherently unsolvable in step 0, $\varphi(r''', W_i) \neq -\infty$ must hold. We let $W_i := W_i \cup \{r'''\}$ and $R := R \setminus \{r'''\}$. If W_i is now such that every DLL is either called by no or by at least two rows, we let $i := i+1$ and goto step 1. Otherwise goto step 2.

Note that this schema gives priority to rows that have not been assigned to a previously computed W_j with $j < i$. It may hence well be, and sometimes is, the case that a perfectly fitting row in such a W_j would make the equation system represented by W_i solvable. The reason for this heuristic is runtime performance: With this approach, the number of fitness computations can greatly be decreased. Otherwise, n fitness computations need to be done for every pick of a candidate row to be included in any W_i which, for large numbers of blocks becomes a practical concern.

The **fitness** of a row r w.r.t. a set of rows W_i is computed as follows. Note that because r may have been chosen from W_i, it is possible that $r \in W_i$. Firstly, let $a = |\{j \in \mathcal{D} : r(j) = 0 \wedge \forall r' \in W_i \setminus \{r\} : r'(j) = 0\}|$ be the number of DLLs that are called neither by r nor by any row in $W_i \setminus \{r\}$. We do not really care about these, but the larger this number, the better. Secondly, let $b = |\{j \in \mathcal{D} : r(j) \neq 0 \wedge \forall r' \in W_i \setminus \{r\} : r'(j) = 0\}|$ be the number of DLLs that are called by r but by none of the rows in $W_i \setminus \{r\}$. These make the row r "inattractive" because we will need to find additional matching rows in further steps, which will potentially lead to large numbers of blocks that need to be hashed together. Thirdly, let $c = |\{j \in \mathcal{D} : r(j) \neq 0 \wedge \exists^{=1} r' \in W_i \setminus \{r\} : r'(j) \neq 0\}|$ count the number of DLLs in W_i that call a DLL that is also called by r and by one row in $W_i \setminus \{r\}$. These DLLs make the row r "attractive" because adding r to W_i reduces the number of DLLs for which no match has been found yet. Finally, if $c = 0$ let the fitness $\varphi(r, W_i) = -\infty$ and otherwise $\varphi(r, W_i) = \alpha \cdot a + \beta \cdot b + \gamma \cdot c$ for suitable parameters α, β, γ. Each W_i then gives rise to one equation system that can be solved in isolation. Note that the above computation amounts to a heuristic breakdown of R into smaller systems without any optimality guarantees.

In our performance experiments, we (somewhat arbitrarily) set $\alpha = 1, \beta = -2, \gamma = 4$ and varied the values of $n \in \{50, 100, 500, 1000\}$ and $d \in \{100, 200, 300, 400\}$. Due to lack of space, we present the results in Table 2 in the Appendix. Note that the time needed by our algorithm increases significantly as d increases, while the increase is not as significant as n increases. This is convenient, since most executables will have a high value for n, while d is much lower, e.g. in our Chromium experiments from §5.1.3, $n = 4749$ and $d = 277$.

3.3.3 *No integer solutions*

It may of course happen that the equation system, or one of the reduced systems, has no non-zero solution, not even a rational one. Let Q' consist of the m non-zero rows of one W_i as computed above. The problem then can be solved by artificially making the equation system $Q'^T \cdot x = 0^m$ underdetermined. To do so, we relax the requirement that the right hand side be 0 and rather leave that open: $Q'^T \cdot x = y$ for an undetermined vector $y \in \mathbb{Z}^m$ (that we construct below; its first $d - m + 1$ entries will be those of vector s below). If $m \leq d$, we simply juxtapose an upper diagonal matrix D of dimension $d \times (d - m + 1)$ to the right of Q'^T and add slack variables $s_1, \ldots, s_{d-m+1}$ to x, resulting in a vector x'. We can then solve $Q'^T D \cdot x' = 0$ which, by construction, is equivalent to solving $Q'^T \cdot x = (s_1, \ldots, s_{d-m+1}, 0, \ldots, 0)$. Because by construction this system is underdetermined, it will have a non-zero integer solution.

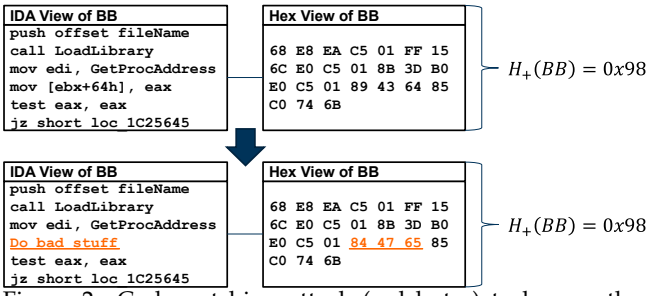


Figure 2: Code patching attack (red bytes) to bypass the checksum function H_+ (byte-wise addition modulo 256)

Intuitively, the above construction of adding columns to Q'^T corresponds to adding rows to Q' which corresponds to adding bogus blocks to the program. These bogus blocks are never executed, because they are placed in dead branches of opaque predicates. However, statically they call the one DLL for which the entry in the respective row of Q'^T is 1.

3.4 Multiple Hash Functions

One attack on this self-checksumming mechanism is patching the code such that the checksum (addition module some number, denoted H_+) is preserved, as illustrated in Figure 2. The idea is to replace original code BB by malign code M and then replace other instructions by inconsequential instructions, such that the checksum of the modified code matches that of the original code. One such inconsequential instruction is `ADD x,x`. When choosing x such that $H_+(\texttt{ADD x,x}) + H_+(M) \equiv H_+(BB) \mod 2^N$, this tampering will not be detected by our algorithm. (Note that H_+ actually is applied to the machine code representation of `ADD x,x`.)

Because of the block structure of our schema, tampering must be performed in place, that is the malicious code M must be smaller than the benign code BB to be replaced.

To counter this attack, we may use a second checksumming function, other than H_+, which includes an operation over the individual machine code bytes. If we want to minimize the number of possible input bytes that result in the same checksum output, i.e. *checksum collisions* we can use a cryptographic hash function. However, computing such a function for every basic block would impose a higher runtime performance impact than using a lightweight checksumming or hashing function. One of the most lightweight checksumming functions is byte-wise XOR-ing (denoted $H_\oplus$). One can verify that byte-wise XOR-ing the machine code bytes before and after run-time memory patching (depicted in Figure 2) results in different checksums. Of course, using both $H_\oplus$ and H_+ together does not guarantee that there will be no collisions. However, it does reduce the probability of a collision.

Note that the above replacement attack as such is difficult (but not impossible) if M contains a call to some DLL. This invariably changes the hash values at runtime. However, the second hash function is necessary because PUPs may also tamper with other checkees and call exactly the same DLLs as M, hence "fixing" the hash value.

Regarding absolute addresses, $H_\oplus$ suffers from the same problems as H_+ described in §3.1. In fact, we are not aware of functions other than byte-wise addition which produce an

invariant checksum over all application restarts. Since x86 instructions are of variable length and are not aligned, at runtime we do not know where an absolute address begins in a sequence of machine code bytes. Therefore, given only the starting address and the size of a contiguous sequence of machine code bytes, one cannot know where the absolute addresses are without additional information or without performing disassembly on the fly, which would have a high performance impact. One could argue that such information is available in the *reloc* section of a PE. However, this would require the checksumming function (which must be lightweight), to look-up whether the current offset it is reading-from contains an absolute address. We consider this far too expensive to be performed before each byte that is read by the checksumming function. Our simple solution therefore is to ignore absolute addresses when using $H_{\oplus}$; they are checked by H_+ only. Absolute addresses are ignored by choosing the checkee blocks (statically, using the x86 Assembly), such that the instructions do not make any references to absolute addresses.

3.5 Cyclic Checks

We inject checks into blocks and want the checkers to be checked themselves. Which blocks check which other blocks is not discussed here; this can be done randomly or be the result of code execution frequency considerations. Let $CB_i = IC_i; BB_i$ be a checker i that combines the integrity check IC_i with the code of block BB_i. Assume that IC_i needs to check the checkee blocks $b(CB_i)$; the identification of these blocks is the result of the computation of a W_i above. Note $b(CB_i)$ may or may not contain CB_i.

Let $b_i = |b(CB_i)|$ be the number of blocks checked by IC_i. The code of IC_i consists of: (1) b_i addresses, sizes and coefficients (computed by solving the equation system corresponding to W_i as constructed above and used as constant factors) of checkees, $CH_{i1}, \ldots, CH_{ib_i}$, each of which differs from the others, (2) one constant hash value for comparison of the aggregate hash values from all checkees,

$$KB_i = \sum_{CB_j \in b(CB_i)} c_{ij} * H(CB_j),$$

which is invariant across different runs of the system (which in turn is made sure by the above construction of linear Diophantine equations; the c_{ij} are the computed coefficients in vector x for the equation system corresponding to W_i); and (3) a pivot X_i, which is a placeholder for a value we need in order to account for cyclic dependencies. We hence have[3]

$$CB_i = CH_{i1}; \ldots; CH_{ib_i}; KB_i; X_i; BB_i.$$

The constant value of KB_i is determined by computing the linear combination of the hash values of the blocks as explained above; remember that the construction of the hash function is such that DLL base addresses are canceled out.

We now need to incorporate hash values for the integrity checks as well. The challenge is to check the integrity if there are cyclic dependencies between checkers, e.g. CB_1 checks CB_2 checks CB_3 checks CB_1. If the code of IC_is are not part of the checksummed bytes, then cyclic checks are not an issue. However, assume IC_is are subject to integrity checking *and* there are cyclic dependencies.

[3]This is slightly misleading because KB_i is a number; think of it as being subtracted from the sum of the hash values and compared to 0.

If $H(A;B) = H(A)+H(B)$, which is the case for the checksum functions we are using, we have

$$H(CB_i) = H(BB_i) + H(X_i) + H(KB_i) + \textstyle\sum_{CB_j \in b(CB_i)} H(CH_{ij}).$$

$H(BB_i)$, $H(KB_i)$ and $\sum_{CB_j \in b(CB_i)} H(CH_{ij})$ are constants. Let G_i denote their sum. We then need to find integer solutions for all X_i such that $H(X_i) = G_i - H(CB_i)$, which is easy to solve if H is addition modulo some number.

4. IMPLEMENTATION

We have implemented the self-checking mechanism for the PE format and the x86 instruction set architecture (ISA). Because the x86 ISA has a variable length encoding where instructions can vary between one and fifteen bytes, byte-by-byte summation modulo 256 (denoted H_+) is the natural checksumming function candidate for satisfying Eq. 1. However, there is an issue with using H_+, which we discuss in §4.2.

4.1 Granularity of Checks

The impact on run-time performance is characterized by the frequency of the checks and the amount of data that is checked. We leave frequency considerations to future work. In terms of the amount of data, checksumming can be applied at various levels of granularity, e.g. memory segments, functions, basic blocks[4], tuples of instructions.

To precompute checksums H_p, we need the compiled and linked machine code. Therefore, we cannot perform the transformation of adding checks at source code level. We have implemented the self-checking transformation such that it is applicable post binary compilation and linking, which also minimizes the impact on the software development process. Because we want to apply our approach to Google Chromium, we use the Google Syzygy Transformation Toolchain as a binary instrumentation framework which is used to post process Chromium (but works for any PE). Among other things, Syzygy performs basic block optimizations via reordering, which improves the cold start times, executable layout and cache efficiency of the instrumented program [14].

Since Syzygy performs a mandatory basic block reordering, the coarsest level of granularity for our transformation is that of *basic blocks*. If we checksum a function composed of several basic blocks, Syzygy will likely reorder its basic blocks, possibly interleaving them with basic blocks of other functions. It is thus not guaranteed that all basic blocks of a function will be placed contiguously in the resulting binary.

4.2 Issues with Byte-by-Byte Addition

In deriving Eq. 1 we used the fact that the difference o between an absolute address a and a base address b is constant and we assumed that the difference between the byte-by-byte summation of these addresses will always be equal to the byte-by-byte summation of the offset o, i.e.,

$$H_+(a) - H_+(b) \equiv H_+(o) \mod 2^N \tag{4}$$

where H_+ is addition modulo some number. This relation is always true for Microsoft Windows PEs that are smaller than 512 KBs because on 32-bit systems all base addresses are 16-bit aligned [27], which means that the base address will always have its least significant 16-bits equal to 0. However,

[4]A *basic block* is defined by a list of sequential instructions ending either with a jump or a return instruction. A basic block has branch-ins only onto its first instruction and branch-outs only from its last instruction.

for PEs larger than $2^{16} \times 8 = 512$ KBs, it could be the case that the base address b plus the offset o leads to a carry bit from one less significant byte to a more significant byte in the absolute address a. For example, consider $b =$ `0x02E00000`, $o =$ `0x00321234`, and $a = b + o =$ `0x03121234`. Relation (4) does not hold for these addresses: $H_+(a) =$ `03` $+$ `12` $+$ `12` $+$ `34` $=$ `5B`, $H_+(b) =$ `02` $+$ `E0` $+$ `00` $+$ `00` $=$ `E2`, and $H_+(o) =$ `00` $+$ `32`$+$ `12` $+$ `34` $=$ `78`; but $H_+(a) - H_+(b) \equiv H_+(o) + 1 \mod 2^N$. The reason is the bit that is carried from the 2nd byte of the absolute address to the most significant byte after adding the offset to the base address.

One intuitive solution would be to simply change the H_+ function such that it does not perform byte-by-byte addition, but word-by-word addition instead. This would solve the issue of the carry bit from one byte of the address to another. Unfortunately, this is not possible because instructions in x86 machine code have variable length, i.e. an instruction can have anywhere between 1 and 15 bytes in length. For example, the *call* instruction from Figure 1 is five bytes long, while the push instruction is one byte long. Therefore, if the H_+ function read one word (four bytes on a 32-bit system) at a time, it would be highly likely that the read words would contain the entire absolute addresses. Moreover, basic blocks have a size equal to a number of bytes divisible by four, which means that reading word-by-word could go outside the boundary of a basic block. Note, however, that word-by-word addition would work on an ISA with fixed width instructions (e.g. ARM, MIPS, etc.).

Another intuitive solution would be to partition the PE into 512 KB blocks and pick checkees and checkers only from within the same partition block, in order to be able to perform strict equality comparison. However, these partitions may not fall at function or basic block boundaries, plus the basic blocks are reordered by Syzygy after the checker to checkee associations are chosen (see §4.1).

Since the base address of a program is chosen randomly by ASLR, we cannot know the exact number of times this carry bit situation could occur. However we can statically compute how many absolute addresses a basic block contains and we can place an upper-bound on how many times this carry bit situation could occur across all possible program restarts (i.e. base addresses). Therefore, for binaries larger than 512 KBs we do not perform an equality comparison of the dynamically computed checksum d to the statically precomputed checksum s. Instead, we check that $|d - s|$ is below the number of absolute addresses in these basic blocks.

Approximately comparing the statically computed against the dynamically precomputed checksum weakens the protection mechanism. Assume an attacker patches a few instructions in protected basic blocks such that their new dynamically computed checksum is equal to d'. If $|d' - s| \leq |d - s|$, then this attack will not be detected by H_+. In §4.4 we add another checksumming function, which further raises the bar for such patching attacks along the lines of §3.4.

4.3 Processing Executable Files

Since a user may not want to protect all functions of an executable, our tool takes as additional input a list of function names to be protected. The tool first disassembles the executable and generates a graph of basic blocks. Second, the following sequence of assembly instructions—the checker—is inserted before each basic block in a function to be protected. These instructions can be obfuscated.

1. Save four values on the stack for each of the N checkees associated with the checker: (i) k_i the coefficients which will be multiplied by the checksum of the checkees, (ii) l_i the number of bytes of each checkee, (iii) a_i the starting address of each checkee and (iv) n_i the number of checkees of each checkee, which are needed for the following reason: each checkee is indicated by an absolute address with the base address equal to the base of the current executable, denoted b. Since the instructions of the checkee also include code that was inserted to perform checks on other checkees, we need to cancel out the absolute addresses of those checkees.
2. Call the checksumming function which reads the machine code bytes at the previously stored addresses on the stack. The checksumming function internally computes the byte-wise addition modulo 256, of the machine code bytes of each checkee. It also computes the *checksum of the base address* for the current binary (denoted $H_+(b)$)[5]. The linear combination of the checksums for all checkees (c_+), which will be the return value of the hash function is:
$$c_+ = \sum_{i=1}^{N} k_i \left(\left(\sum_{j=0}^{l_i - 1} \mathcal{M}[a_i + j] \right) - n_i H_+(b) \right) \mod 256,$$
where $\mathcal{M}$ denotes an array representing the process memory of the protected application. The intuition as to why absolute addresses are canceled out by this linear combination of checksums is given in §3.2.
3. Compare the value returned by the checksumming function (i.e. c_+) with a hard-coded pre-computed value.
4. Trigger the response functions if the values in the previous comparison are not equal, otherwise continue the normal execution of the original binary.

Thirdly, the basic blocks are reordered by Syzygy and laid out in the output executable. This reordering changes several relative and absolute addresses. In the final step, our tool hence patches the precomputed checksums (in-place) such that they will match the dynamically computed checksums.

4.4 Detecting Checksum Collisions

To detect patching attacks which result in checksum collisions we resort to using $H_\oplus$ only over consecutive instructions which do not contain absolute addresses (§3.4). We call them *chunks*. This checksumming function is used in addition to H_+ for basic blocks that contain absolute addresses. Note that we can use any checksum or hash function to hash such code chunks; we chose $H_\oplus$ because it has a low performance overhead. Technically, we consider chunks of code to be any maximum subset of consecutive instructions inside of a basic block that do not contain references to absolute addresses. For example, the basic block from Fig. 2 contains 2 chunks: the first consists of only the first instruction, and the second consists of the last 3 instructions.

For each of the M chunks associated with it, the checker saves the following values on the stack: (i) l_i the size in bytes of the chunk and (ii) a_i the starting address of the chunk. Let $c_\oplus$ denote the return value of the function:
$$c_\oplus = \sum_{i=1}^{M} \left(\bigoplus_{j=0}^{l_i - 1} \mathcal{M}[a_i + j] \right) \mod 256.$$

[5]This base address is known because (in the implementation of H_+) we inserted a *mov* instruction that copies the value of a reference to the base of this binary in an auxiliary register, which it then dynamically hashes.

The output of this $H_{\oplus}$ checksum function is combined with the result of the H_{+} checksum function: $c_{+} + c_{\oplus} \mod 256$. The resulting value is hard-coded in the protected PE and is compared against the checksum computed at runtime. If the comparison fails, the response function is called.

4.5 Response Functions

Response functions ideally are (1) *transparent*, i.e. not noticeable to the end-user. They must be effective against attacks and should be (2) *stealthy*, i.e. an attacker should not be able to identify and disable the response. We are not aware of response functions which are both transparent and stealthy. However, we did implement one transparent and one stealthy response function.

The transparent yet non-stealthy response function consists of sending a signal to a trusted server indicating a change in the process memory. This can only be achieved by using a system call that accesses the OS network interface.

The stealthier response function crashes the program after tampering is detected. Crashing could be implemented in a variety of ways, i.e. instead of comparing the precomputed hash to a hard-coded checksum, we can use the hash to compute a jump target or perform an operation with the stack pointer. This may not crash the program immediately, but eventually will lead to a crash with high probability.

4.6 Limitations

Currently we assign checkees randomly to checkers with the constraint that a linear combination of their hashes is invariant to application restarts. However, to reduce performance impact this random approach can be improved to always pick hot code as checkees and less hot code as checkers.

Due to the way in which self-checksumming works, it causes issues during debugging. Generally debuggers (e.g. the Microsoft Windows debugger, GDB) replace the first byte of a basic block with `CC`, which is the opcode for the *int 3* assembly instruction. This is the standard way for debuggers to gain control, from the application that is being debugged. After the debugger gains control it changes the `CC` value back to its original value. Therefore, if the application that is being debugged employs self-checksumming, then it could happen that a checker computes a checksum over a basic block where the debugger has changed the first byte to `CC`. This will lead to the wrong checksum value which will trigger the response function during debugging.

A solution to this issue is to use *hardware breakpoints* if the debugger supports them. These kinds of breakpoints do not cause the debugger to replace the first byte of a basic block with `CC`. The drawback to this solution is that the number of hardware breakpoints on the x86 architecture is limited to 4.

5. EVALUATION

5.1 Performance Evaluation

We evaluated the performance of our implementation over two categories of applications: (1) CPU-intensive applications such as an XML parser, which indicates the worst-case performance overhead due to a higher frequency of checksumming, and (2) GUI applications (that are less CPU-intensive) such as the Chromium browser, which involve continuous human-user interaction and indicate an average performance overhead for self-checksumming protection.

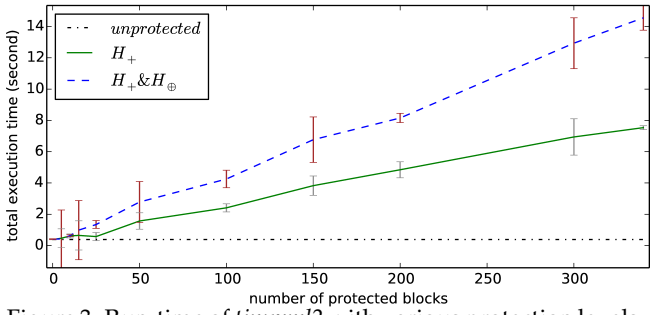


Figure 3: Run-time of *tinyxml2* with various protection levels.

5.1.1 XML Parser

We selected *tinyxml2* [34] from Github's trending repositories as the first candidate for our performance evaluation. It is an open source XML file parser library that reads an input file and constructs the corresponding Document Object Model (DOM). The library is written in C++ and consists of 4571 lines of code. The compiled binary consists of 340 basic blocks and occupies 136 KB of disk space.

To measure the overhead of our implementation, we selected ten random combinations of 5, 10, 15, 20, 50, 100, 150, 200, 300 and 340 basic blocks, and used them as input configurations for our tool. For each input configuration we generated two protected binaries: one where only the H_{+} checksum was used, and one where both H_{+} & $H_{\oplus}$ were used. This resulted in $10 \times 10 \times 2 = 200$ protected binaries.

These protected binaries were executed with the same input XML file set shipped with the library, which varies in size from 0 KB (empty test) up to 142 KB (`dream.xml`). We measured the total process run-time as the difference between the timestamp when the process terminated its execution and the time when execution was started. The total process run-time was measured 100 times for every application. The mean and standard deviation were computed across all applications that have the same number of protected functions, in order to weed out random influences on runtime performance. Figure 3 illustrates the evaluation results for protection with H_{+} and with H_{+} & $H_{\oplus}$. The baseline execution time of 0.4 seconds (y-axis) can be seen for zero protected blocks (x-axis). Using H_{+} slows down execution about 20 times when all 340 blocks of the application are protected, while the slowdown induced by H_{+} & $H_{\oplus}$ it about 35 times. This slowdown may be unacceptable in scenarios which require high-responsiveness or high-throughput. However, it may be acceptable in scenarios where behavior integrity of some software functions (e.g. license checks), is of utter importance.

5.1.2 Chromium

Chromium involves user interaction and therefore is not as CPU intensive as the XML parser. By measuring the total process run-time of versions of Chromium protected using different input configurations, we could not observe a slowdown compared to the unprotected version. Another reason for this low overhead is the multi-process and multi-threaded architecture of Chromium, where several processes depend on input values gathered via OS system calls (e.g. network communication) and are not controlled by the end-user inputs. Therefore, we measured the overhead incurred per protected function, i.e. the relative increase in run-time of a protected function w.r.t. its unprotected counterpart.

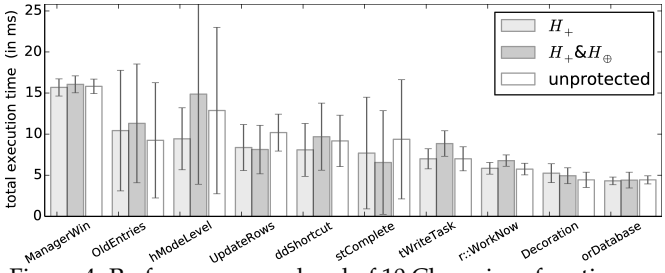


Figure 4: Performance overhead of 10 Chromium functions.

Protection	Mean	Std. Dev.	Median	Maximum
H_+	1.34	1.16	1.00	8.2
H_+ & $H_\oplus$	1.34	0.89	1.06	5.0

Table 1: Relative performance overhead in ms (protected function run-time divided by unprotected function run-time) of self-checksumming protection in Chromium.

One way of measuring the overhead per function is using a binary profiling tool such as *SyzyProf*, offered by the Syzygy toolchain. However, such tools also require instrumenting the application. Applying *SyzyProf* on an application transformed by our tool breaks the precomputed checksums due to the profiling instructions that *SyzyProf* adds to the code. Applying *SyzyProf* first and then transforming it using our tool solves this issue, however, it causes the profiling information collected by *SyzyProf* to be inaccurate, because code is inserted and reordered by our tool.

We therefore extract the profiling information from a protected version of Chromium via the browser itself using the chrome://profiler URL. Before the profile information is saved we let the browser run for 30 seconds. During this time it loads its homepage (i.e. https://www.google.com), then we search for the word "hello". After the search results are loaded we open the chrome://profiler. From the profile information we compute the average execution time per function by dividing the total execution time and the number of times a function was executed, for each of the protected functions. Similarly to the XML parser, we protect Chromium both with H_+ only and with both H_+ & $H_\oplus$.

We have protected 400 functions from Chromium, 73 of which were executed and consistently appearing in the profile information across 20 executions of Chromium. Looking at the names of the remaining functions, we realized that they are only called when performing other end-user actions, e.g. changing browser preferences, downloading files, etc., which we did not perform in our test runs. Table 1 shows the average overhead over these 73 functions protected with H_+ only and protected with H_+ & $H_\oplus$. The mean values are the same for both, however, the median is larger when both H_+ & $H_\oplus$ are used. Remarkably the maximum overhead was observed for a function protected with H_+. Figure 4 shows a bar chart with the average execution times and the standard deviation of 10 randomly chosen functions from Chromium protected with H_+ only, protected with H_+ & $H_\oplus$ and unprotected. From the bar chart we can see that the impact of self-protection is relatively moderate. Also it seems that for some functions the performance impact of protection with H_+ is lower than with H_+ & $H_\oplus$. This is counter-intuitive, however, we are not certain that the inputs for these functions during different runs are the same, due to the previously mentioned reason, i.e. Chromium performs several system calls, whose execution times vary depending on the OS state.

5.1.3 Protection Time and Protected Binary Size

The time required to protect an executable, i.e., to statically add the checking code, increases linearly with the number of basic blocks to be protected. For instance, protecting 400 functions with 4749 basic blocks from Chromium, takes around 27 minutes, out of which 14 minutes are used only by the Syzygy binary disassembler, which is performed before our protection mechanism is applied. On the other hand, protecting 5 functions with 49 basic blocks from Chromium, takes around 15 minutes, out of which 14 minutes are still required by the binary disassembler stage. Therefore, the disassembly stage is constant for a certain executable file, while our protection mechanism increases linearly with the number of protected basic blocks.

The size of the protected executable is increased w.r.t. the unprotected executable. Similarly to protection time, the size of the protected executable also increases linearly with the number of protected basic blocks. For each protected basic block the added integrity check code with H_+ is at least 82 bytes, when the added code is not obfuscated via *superdiversification* [18]. Note that this integrity check has two checkees and the size of the machine code that pushes the information on the stack for each checkee is 20 bytes per checkee. Adding also $H_\oplus$ also requires pushing the information for four chunks and some extra instructions to call the $H_\oplus$ hash function, save its result and combine it with the result of the H_+ hash function. The size of the machine code that pushes the information on the stack for each chunk is 15 bytes per chunk. Therefore, the total size of the resulting integrity check code is 163 bytes (unobfuscated) per basic block, when both H_+ & $H_\oplus$ are used.

5.2 Security Evaluation

One intuitive way of evaluating the security of our approach is to develop or use an existing PUP which tampers with the process memory of the application. However, we argue this is not a fair evaluation since such a browser hijacker would not be aware of the protection mechanism we are employing and would therefore be detected.

An alternative way of performing the security evaluation is to assume that the attacker is fully aware of all details of our protection mechanism and consider possible actions of such an attacker. The attack tree in Figure 5 on the next page shows possible attacks which an attacker may implement to bypass our protection mechanism. The root of the tree shows the goal of the attacker and each of the leaves are alternative ways to achieve this goal. The following subsections discuss each of the leaves of this attack tree.

5.2.1 Disable Checker of Targeted Code Bytes

Each basic block in a protected function is checked by at least one other basic block located in a different function. This causes cyclic dependencies between checks which are discussed in §3.5, resulting in a connected directed graph where each node is a basic block and each arc indicates the source node performing an integrity check over the destination node, illustrated in Figure 6a.

Additionally, each basic block in the protected executable also checks a number of chunks spread across different basic blocks, which adds another kind of directed arcs (showed

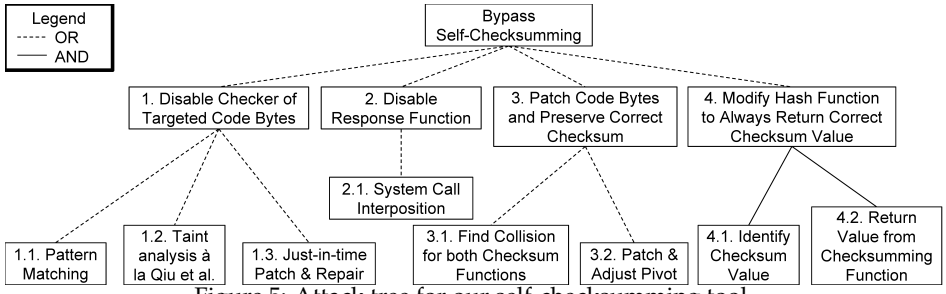
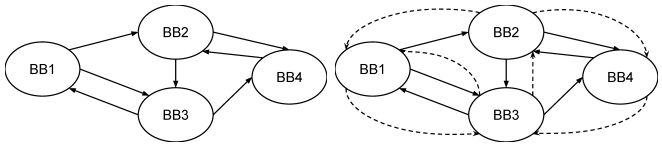


Figure 5: Attack tree for our self-checksumming tool.



(a) With H_+ (b) With $H_+ \& H_\oplus$

Figure 6: Connected graphs of basic blocks, i.e. cyclic checks.

with dashed lines in Figure 6b) to the previously mentioned graph. Dashed lines also indicate that $H_\oplus$ is not computed over the entire destination basic block but only on a part of it (i.e. a chunk). This strongly connected graph forces the attacker to *identify and disable all checkers* before being able to patch any of the basic blocks in the executable.

The attacker could perform a pattern matching attack to identify all checkers inside a protected executable and disable them by replacing their code by NOP instructions. In order to avoid patterns in the integrity check code, we employed an obfuscation transformation known as *superdiversification* [18], i.e. replacing one sequence of instructions in a basic block by another sequence of instructions which is functionally equivalent. The drawback of employing this kind of obfuscation is that the size of the integrity checking code, which is added to each protected basic block, is increased 2-4 times relative to the current size of the code (see §5.1.3).

Another way of detecting and disabling checkers was presented by Qiu et al. [28]. They taint the code bytes of an executable and perform backward and then forward taint analysis to identify checker instructions, which use the (tainted) code as data for branch decisions. Together with the authors of [28], we employed their tool on an execution trace generated by a protected version of Chromium running for 1 minute. The tool executed on machine with 32GB of RAM and an Intel Xeon CPU with 2.0 GHz clock speed and was able to process 1% of the trace per day. We conclude that this attack is effective, however, it does not scale for applications which generate large execution traces.

The time interval (*detection interval*) between a patching attack and detection of the patch via our self-checksumming mechanism can vary greatly due to the random way in which checkers and checkees are picked, i.e. if a basic block executed rarely is checking another basic block which is patched, then the detection interval will be relatively large compared to the scenario where the basic block which is performing the check would be executed frequently. Checkers could be bypassed if the attacker patches the targeted code before it executes and reverts the patch after it executed and before any checksum is performed on it. We call this *just-in-time patch & repair*. However, due to Chrome's multi-process and multi-threaded architecture it is highly unlikely that this attack could succeed reliably across a large number of end-users, due to the unpredictability of the OS scheduler [7]. In order to minimize the detection interval for more security sensitive code we propose using heuristics based on profiling information of the protected application. This way, security sensitive code would be checked by multiple integrity checks in frequently executing code. We leave this straightforward extension of our implementation for future work.

5.2.2 Disable Response Function

A process running under the same privileges as the target application process (which is the case for PUPs) can intercept all system calls and prevent the transparent response function from sending any information to a trusted server. This means that even if there were multiple response functions obfuscated and spread-out across the code of the PE, attacks such as system call hooking could still detect and stop such response functions.

An attacker would need to employ the techniques of §5.2.1 to identify the stealthy response function. A delayed crash is even harder to trace back to its root cause by an attacker.

5.2.3 Patch Code Bytes and Preserve Checksum

This attack was described in §§3.4 and 4.4. The idea is to patch a certain number of bytes in the machine code such that the checksum is preserved, as illustrated in Figure 2. To reduce collisions we added the $H_\oplus$ checksum function over chunks. This does not completely exclude the possibility of a collision, however, it decreases its likelihood.

Another attack is to modify bytes in the code even if the checksum is not preserved and then patch the pivot byte such that the checksum is corrected. This is detected by checking the chunk containing the pivot byte.

5.2.4 Modify Hash Function to Always Return Correct Checksum Value

An attacker who knows our self-checksumming implementation may attempt to modify the machine code of the checksumming function such that it always returns the correct checksum. This is possible by reading bytes starting from the return address of the checksumming function until a comparison instruction is first encountered (the check if the statically and dynamically computed hash values are identical). The attacker could then take the constant value from this instruction and return it from the checksumming function. We can raise the bar against this attack by employing superdiversification [18], i.e. use different types of instructions to compare the return value of the checksumming function with the precomputed checksum in different checkers.

6. CONCLUSIONS AND FUTURE WORK

We have presented the design, implementation and evaluation of a mechanism for software tamper detection via self-checksumming. The results indicate that for CPU-intensive applications, the run-time of an application having all functions protected may be 20 times as high as the unprotected application when using one checksumming function, and up to 35 times higher when using two checksumming functions. This kind of overhead may appear prohibitive for some applications. However, we note that an application developer may not necessarily want to protect *all* functions of an application, which significantly lowers the overhead. Moreover, the performance of a protected application depends on how frequently checks are executed. If checks are inserted inside frequently executing code such as deeply nested loops, then performance impact is higher than if checks are inserted in less frequently executed code.

For non CPU-intensive multi-process and multi-threaded applications such as GUI applications, the performance impact experienced by the end-user may be acceptable, which is the case in our Chrome case study.

Obfuscating the instructions of the inserted checks increases the security, but degrades the performance and increases the size of the protected binary. This makes application of self-checksumming practical for less CPU-intensive applications and in situations where only a subset security critical set of the functions of the PE are protected.

One possible direction of future work is protecting the information from data blocks. This feature would protect hard coded values such as URLs and pointers to functions in other libraries. Another direction of future work is using profiling information from the original (unprotected) PE, to create groups of basic blocks or functions according to their execution frequency. This would enable picking checkees as hotter code and checkers as less hot code. This feature would reduce the performance overhead of the protected binary.

7. REFERENCES

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[2] S. Banescu, A. Pretschner, D. Battré, S. Cazzulani, R. Shield, and G. Thompson. Software-based protection against changeware. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 231–242. ACM, 2015.

[3] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[4] B. Blietz and A. Tyagi. Software tamper resistance through dynamic program monitoring. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3919 LNCS:146–163, 2006.

[5] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2001.

[6] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2002.

[7] A. Colesa, R. Tudoran, and S. Banescu. Software random number generation based on race conditions. In *Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC'08. 10th International Symposium on*, pages 439–444. IEEE, 2008.

[8] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 319–328. ACM, 2012.

[9] P. Dewan, D. Durham, H. Khosravi, M. Long, and G. Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring simulation multiconference*, pages 828–835. Society for Computer Simulation International, 2008.

[10] Evenbalance. PunkBuster — Online Countermeasures, 2015. http://www.evenbalance.com/pbsetup.php, [Online; accessed 20-September-2016].

[11] W.-c. Feng, E. Kaiser, and T. Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20. ACM, 2008.

[12] T. Garfinkel et al. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, volume 3, pages 163–176, 2003.

[13] 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.

[14] Google. Syzygy – profile guided, post-link executable reordering, 2013. https://github.com/google/syzygy/wiki/SyzygyDesign, [Online; accessed 12-September-2016.

[15] G. Hoglund. Hacking world of warcraft: An exercise in advanced rootkit design. *Black Hat*, 2006.

[16] 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, 2001.

[17] Intel. Intel Active Management Technology — Query, Restore, Upgrade, and Protect Devices Remotely, 2016. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html, [Online; accessed 20-September-2016].

[18] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer, 2008.

[19] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[20] M. H. Jakubowski, C. W. N. Saw, and R. Venkatesan. Tamper-tolerant software: Modeling and implementation. In *International Workshop on Security*,

pages 125–139. Springer, 2009.

[21] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*, pages 3–9. IEEE Press, 2015.

[22] E. Kaiser, W.-c. Feng, and T. Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 269–279. ACM, 2009.

[23] S. Mansfield-Devine. The promise of whitelisting. *Network Security*, 2009(7):4–6, 2009.

[24] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6201 LNCS:21–40, 2010.

[25] W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 82–89. ACM, 2007.

[26] B. Morgan, E. Alata, V. Nicomette, M. Kaâniche, and G. Averlant. Design and implementation of a hardware assisted security architecture for software integrity monitoring. In *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, pages 189–198. IEEE, 2015.

[27] M. Pietrek. Peering inside the PE: a tour of the win32 (R) portable executable file format. *Microsoft Systems Journal-US Edition*, pages 15–38, 1994.

[28] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. Identifying and understanding self-checksumming defenses in software. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218. ACM, 2015.

[29] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys (CSUR)*, 49(1):4, 2016.

[30] 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*, 2005.

[31] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[32] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *2015 IEEE Symposium on Security and Privacy*, pages 151–167. IEEE, 2015.

[33] K. Thomas, J. A. E. Crespo, R. Rasti, J.-M. Picod, C. Phillips, C. Sharp, F. Tirelo, A. Tofigh, M.-A. Courteau, L. Ballard, et al. Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software. In *USENIX Security Symposium*, 2016.

[34] L. Thomason. TinyXML2. https://github.com/leethomason/tinyxml2, 2016. [Online; accessed 20-September-2016].

[35] Valve. Valve Anti-Cheat System (VAC), 2015. https://support.steampowered.com/kb_article.php?p_faqid=370, [Online; accessed 20-September-2016].

[36] B. Wyseur. White-box cryptography. In *Encyclopedia of Cryptography and Security*, pages 1386–1387. Springer, 2011.

[37] Y. Younan, W. Joosen, and F. Piessens. Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures. In *Technical report, KU Leuven*, 2004.

APPENDIX

Blocks vs. DLLs	100	200	300	400
50	0.394	0.481	0.676	1.240
100	0.580	1.105	1.317	1.086
500	3.045	223.579	6299.026	14894.043
1000	7.057	376.748	9023.435	21225.647

Table 2: Time (in seconds) needed by *equation system reduction algorithm* (presented in Section 3.3.2), to output the solution (including equation solving by the SMT solver), corresponding to randomly generated Q matrices having n blocks (indicated in the first column) and d DLLs (indicated in the first row). Note that the performance numbers may vary depending on the actual values inside of the Q matrix and the random seed used by the SMT solver.