A Graph Game Model for Software Tamper Protection

Nenad Dedić *, Mariusz Jakubowski **, and Ramarathnam Venkatesan ***

Abstract. We present a probabilistic program-transformation algorithm to render a given program tamper-resistant. In addition, we suggest a model to estimate the required effort for an attack. We make some engineering assumptions about local indistinguishability on the transformed program and model an attacker's steps as making a walk on the program flow graph. The goal of the attacker is to learn what has been inserted by the transformation, in which case he wins. Our heuristic estimate counts the number of steps of his walk on the graph. Our model is somewhat simplified, but we believe both the constructions and models can be made more realistic in the future.

1 Introduction

In this paper, we consider the problem of protecting a complex program against tampering. The results of [3, 11] mean that we cannot hope to solve this in general, namely in a model involving worst-case programs and polynomial-time adversaries. Hence it is natural to ask for practical solutions in some natural model with limited attacks. Here the hard problem is in building an appropriate model. A careful look at well known attacks (see overview in Section 3) and effects of program-transformation tools on local program properties (e.g., how homogeneous the code looks over 50 lines of assembly code) allows us to propose a security model of various protection schemes, based on some assumptions.

Our overall approach is as follows. First, we take a given program P and convert this into another program P', where we inject new code that modifies the control and data flow graphs by adding nodes and edges. The goal of the attacker is to find the new additions, and we would grant the attacker victory if he does this reliably. Thus, we specify a formal model by defining a game where the attacker's moves correspond to various attempts to break the protection, and the attacker's victory corresponds to a break. In the present state of software protection, models based on complexity theory offer mainly negative results [3, 11], with a handful of positive results that essentially formalize hash-based comparisons [12, 18]. Motivated by an assortment of heuristic techniques for tamper protection, we give a simplified model, which captures realistic scenarios and allows quantitative analysis of tamper-resistance. Our model makes

* Computer Science Department, Boston University. nenad@cs.bu.edu. Part of this work was done during internships at Microsoft Research (Redmond, WA).
** Microsoft Research (Redmond, WA). mariuszj@microsoft.com
*** Microsoft Research (Redmond, WA and Bangalore, India). venkie@microsoft.com

engineering assumptions about local indistinguishability of small code fragments and provides a lower bound on the attack effort required.

An adversary who tries to reverse-engineer and eventually "crack" the program is typically equipped with some software tools that allow him to analyze the static structure of the program, and execute it in some controlled way. It seems very difficult to design a scheme that transforms *any* program into a semantically equivalent one, but which is protected against malicious changes.

2 Our Approach

The rest of this article is structured as follows, with a main goal of motivating the model and an algorithm for protection.

1. An overview of known tamper-protection techniques and attacks.
2. A randomized algorithm for tamper-proofing complex programs.
3. A graph-game-based model of an attacker's interaction with programs.
4. Lower bounds on an attacker's resources to break the protected program.

Our approach involves inserting k *local tamper-detection checks* in the program. Each check is responsible for detecting tampering in a small portion of the program, consisting of s program fragments. At least f checks are required to fail before a tamper response is triggered, thus we have a *threshold tamper response*. Next we make use of *homogenizing transformations*. They a given program into a semantically equivalent one, but such that local observations (those confined to instruction sequences of length at most b) can be assumed to reveal no useful patterns.

Neither of the above methods offers sufficient security if used alone. The task of our algorithm is to inject the local checks in a randomized way, create the threshold tamper response and perform program homogenization.

We then show a model which we believe captures most of practical attacks against our method. This model does not necessarily cover tamper-protection schemes based on different ideas. Finally, relying on certain conjectures, we show some lower bounds on attacks. Some aspects and components of our algorithm have been implemented and studied in practice for the viability of our models and assumptions. This article presents a first effort in formalizing them. We believe that our methods can be further refined, and heuristic estimates on attack complexity can be made more realistic.

3 Some Typical Protection Schemes and Attacks

Our model is aimed at capturing most practical attacks. Its viewpoint involves forcing an attacker into learning and playing a graph game, whose winning strategy has a lower bound in the number of game steps under suitable assumptions.

These come down to some engineering assumptions about certain code transformations, which we believe can be made to hold with further research. To justify this model and provide context, we survey prevalent tamper-protection techniques and attacks. We start from the easiest, which offer least security, and proceed to more sophisticated ones. We denote the program to be protected by P and the attacker by A.

3.1 Single-point license check

P is protected by adding a subroutine L which verifies some condition, such as a correct digital signature validating its authenticity or a license. L is called from other parts of P, and normal operation is resumed only if L returns "true".

Attack. Using simple control flow analysis, A can identify L. A then patches L to return only "true".

3.2 Distributed license check

To thwart the previous attack, L is broken into pieces or copied with some variations, additionally obfuscated, which are spread throughout P.

Attack. A can make a preliminary guess for the location of one copy of L. A's goal now is to find the locations where variations of L or its components may be scattered. Robust binary matching tools such as [17] can be used to identify other copies of L. Other attacks use flow graph analysis. That is, A computes the flow graph of G. The guessed copy of L induces a subgraph H. A copy of L elsewhere in the program induces a subgraph similar to H, and it can be found via subgraph embedding. Another flow graph based attack uses the fact that typically, the code corresponding to L is a component which is weakly connected to the rest of G. After identification, A can patch the calls to L. These attacks are considerably advanced in comparison in terms of the tools needed to implement them in practice.

3.3 Code checksums and integrity-verification kernels

Tampering can be detected during runtime by loading code segments, and computing checksums. P runs only if they agree with precomputed values [2].

Attack. The task of loading a code segment for reading is an unusual occurrence in typical programs, and can be trapped. Using some hardware support, more generic attacks on code-checksum schemes are described in [15]. Unusual execution patterns (read accesses to code segment, paging faults) can be exploited.

3.4 Oblivious hashing

Consider a program fragment F that uses some set of variables X. For an assignment x to variables X, *execution trace* $e(x)$ is the sequence of all values of X during execution of $F(x)$. Oblivious hashing [4] is a method whereby for a subset of variables $Z \subseteq X$, $F(x)$ produces a hash $h_Z(e(x))$. For suitably chosen random inputs r, tampering with values of Z during execution of $F(r)$ will produce e' for which $h_Z(e') \neq h_Z(e(r))$ with high probability. OH can be used to detect code tampering. It is resistant to the attacks of [15] because the code segment is never read or used as data, and calls used to compute OH cannot be easily separated from regular calls.

Attack. If correct values of $h(e)$ are precomputed and stored for test inputs, then A may discover these values, since they may look special or random (see Subsections 3.8, 3.9). If one uses $h(t)$ indirectly (to encrypt some important variables) or computes $h(t)$ during runtime using a duplicate code segment, then without sufficient precaution, A can attack via program analysis. Methods for addressing these attacks are discussed in this paper.

3.5 Anti-disassembly and diversity

Here an attacker converts a released executable into assembly so that it can be understood, using a disassembler which itself may have built-in graph-analysis tools [7]. A defense would be to cause incorrect disassembly by exploiting different instruction lengths to cause ambiguity involving data and code. At best one can cause a handful of attempts to disassemble, but it is unlikely using only these that one can force a significant number of runs of the disassembler. More flexibility is offered by virtualization and individualization [1], where the idea is to force A into learning a new virtual machine (VM) for attacking each copy. An instance I of the program P is implemented as a (V_I, P_I), where V_I is a virtual machine and P_I is code which implements P under VM V_I. To execute the instance I, one has to run $V_I(P_I)$. Disassembling is difficult because A does not have the specification of V_I. Furthermore, even if A disassembles (V_I, P_I), this is of little help in disassembling (V_J, P_J) for $I \neq J$, because of instance randomization.

Attack. The scheme is open to attacks which do not rely on detailed understanding of P_I. In a *copy attack*, A tampers with P_I and then learns which code is responsible for the resulting crash: It saves program state s before some suspect branch, and tries multiple execution paths from s. If most of these paths end up crashing in the same place, then it must be some previous branch that is causing the crash. A can now make a new guess and repeat the attack.

3.6 Defense against copy attacks

A defense against the above attack involves *distributed tamper-detection checks* and *threshold tamper-protection scheme*. Distributed tamper-detection check is embedded in s program fragments $F_1, \ldots, F_s$, and it has a chance p of failing if each of $F_1, \ldots, F_s$ is tampered with. To disable a check, the attacker must identify all s code fragments $F_1, \ldots, F_s$. Threshold tamper-protection scheme embeds k checks in the program, and it causes the program to crash (or initiates some other security response, such as performance degradation or disabling features) only after at least f checks fail.

These two techniques make it difficult for the attacker to locate the protection scheme without detailed examination of the code and careful debugging. Additional ideas can be used to increase the security, for example delaying the crash even after f checks have failed.

Attack. A can reduce size of the search space by using control- and data-flow analysis.

3.7 Program-analysis tools

Against static control-flow analysis, the idea is to make the control-flow graph look like a complete graph. Computed jumps and opaque predicates can be used

for this. A computed jump explicitly calculates the jump-target address, and this calculation can be obfuscated. Similarly, an opaque predicate [6] calculates the value of a predicate in an obfuscated way. Sufficiently strong obfuscation can reduce the usefulness of control-flow analysis.

Against data-flow analysis, the idea is to make the dependency graph of k variables look like a complete graph on k nodes. A should gain no useful information about dependencies of those variables. Lightweight encryption (LWE) can be used for this.

3.8 Unusual-code-detection attacks

Certain transformations used in program protection can introduce unusual code patterns. For example XOR and other arithmetic instructions are less often used, but a protection mechanism or (light-weight) encryption may use them often, where they can be spotted by localized frequency counts of such opcodes. Semantics-preserving peephole transformations or adding chaff code can be used to make code appear more uniform. Iteration and randomization can be used to diffuse well.

3.9 Randomness detection attacks

Some protection mechanisms may embed encrypted code segments, which may make them vulnerable to attacks of [13]. This attack was designed to find high-entropy sections, such as cryptographic keys that may contain 1024 bits. To prevent code attacks, near-clear encryption of code may be used (by transforming a code fragment into another one that still looks like valid code). Protection of data segments against such attacks involves keeping all data in an encrypted-randomized form.

3.10 Secure-hardware oblivious execution of arbitrary programs

In the scheme of [10], P is converted into P' whose data access pattern is completely random. In each step, a fresh random address is accessed, and a random value is written to it. The scheme offers very good security guarantees, but is impractical. For a program of size n, it suffers a $\log^2(n)$ overhead in running time, and because of random data access, locality of reference is lost.

By relaxing the notion of obliviousness [16] to an attacker's inability to narrow down the location of a variable observed in memory location at time $t = 0$ after $t = T$, one may restore locality of reference for some data-structure operations. This oblivious data structure also requires hardware, but may be simulated by software in practice. While this opens new attacks, defenses may use all the methods discussed in this paper, enabling a modular approach to designing protection systems.

4 The Protection Algorithm

At a high level, our protection scheme works as follows. Let P be the program which we wish to protect. Suppose that there is some programmer-specified *critical code* L. L returns a boolean output, indicating some condition required for proper program execution (such as validity of a license). However, L need not

implement any otherwise useful functionality of P. Our goal is to link L and P so that P executes properly only if L returns "true" despite tampering attacks. The protection algorithm proceeds in phases:

Critical code replication and embedding. L is replicated into l copies $L_1, \ldots, L_l$. Each copy L_i is embedded into P, so that if L_i returns "false", then P is corrupted or terminated. This phase could require significant manual intervention: Specifying suitable points where L_i can be embedded could require programmers' insight into code.

Graph transformation. P is transformed suitably, so that we can assume that its flow graph can be adequately modeled as a random regular graph, and program executions look like random walks.

Check insertion. k checks $C_1, \ldots, C_k$ are randomly inserted into P. Each C_i locally checks for untampered execution.

Creating dependencies. P is transformed so that it crashes when a subset of f checks fails.

4.1 Primitives

We base our scheme on the existence of certain primitives. No single primitive suffices to achieve security against tampering. They must be used in conjunction to ensure adequate protection, and a meaningful model in which a security analysis is possible. Ideas that demonstrate plausibility of those primitives will be briefly mentioned, but a detailed discourse is beyond the scope of this paper. We believe that it is possible, with sufficient research effort, to implement them with satisfactory security.

Flow graph transformation. Our protection algorithm requires some way to change P into a functionally equivalent Q whose flow graph G has the following property. An execution of P cannot avoid any substantial (i.e., constant) fraction of nodes of G for too long (except with small probability). The precise formalization and quantification of these statements depend on the system parameters and desired security level, and some analysis will be given later in the text.

Roughly, we will assume that G can be modeled by an expander graph, and that P's execution resembles a random walk on G as much as possible (even when tampered by the attacker). We assume that there is a function $(V, E) = \mathsf{GraphTransform}(P, n)$ that returns a program Q whose flow graph is a good expander graph of n nodes with the above properties.

This transformation can be approximately achieved by combining various obfuscating transformations, such as code replication, diversification and overlapping [1]; opaque constructs [6]; and data and control-flow randomization [5].

We will assume that the program is partitionied, where the partion is given by sets of flow graph nodes $F_1, \ldots, F_n$. By picking n properly and assuming the sizes of F_i's are approximately the same, we will assume that the flow graph induced by this partition is an expander graph.

Checks. A check C is specified by s code fragments $F_1, \ldots, F_s$. If some F_i is tampered, then with probability $\mathsf{p_{detect}}$ it will be *triggered*. If all of $F_1, \ldots, F_s$ are triggered, then the check C *fails*. We assume that the attacker cannot prevent the check failing, unless he identifies all $C_1, \ldots, C_s$. Let $C = \mathsf{InsertCheck}(F_1, \ldots, F_s)$ denote the function that produces a check C given code fragments $F_1, \ldots, F_s$. A brief outline of a possible check implementation follows.

Tampered code can be triggered using integrity verification kernels (IVK). If IVKs are implemented using oblivious hashing, then runtime tampering is detected and registered as an incorrectly computed hash value. For example, to insert a triggering mechanism in F_i, one embeds oblivious-hash computation in F_i, resulting in a hash value h_i. Some correct values h_i^* can be precomputed (or h_i can be compared against the hash of a copy of F_i elsewhere in the program). Suppose correct h_i^* are precomputed. To ensure that C fails only after all of $F_1, \ldots, F_s$ are triggered, some other code could compute the product $(h_1 - h_1^*) \cdot \cdots \cdot (h_s - h_s^*)$. This product can be nonzero only if all the hash values are incorrect. Product computation should, of course, be obfuscated. Appropriate action can be taken, depending on whether the product is zero. [4, 2]

Check detection-response. We assume that check failures can be detected, and a response can be effected after some programmer-specified subset of checks has failed. More precisely, suppose k checks $C_1, \ldots, C_k$ are installed. The programmer will specify the *response structure* – a set R whose elements are subsets of checks (so R contains elements x, where each $x \subseteq \{C_1, \ldots, C_k\}$). The response will be triggered only upon failure of a subset of checks $y \subseteq \{C_1, \ldots, C_k\}$, such that $x \subseteq y$ for some $x \in R$.

For a detailed account on a possible check response implementation refer to [14]. We note here that their implementation provides for tampering response after a specified number f of checks fail. Let us denote the corresponding response structure by $R_f(C_1, \ldots, C_k)$. Then $x \in R_f(C_1, \ldots, C_k) \iff x \subseteq \{C_1, \ldots, C_k\} \wedge |x| = f)$. A practical advantage of their approach is is that the response is well separated both temporally and spatially from the checks that cause it.

We will denote with $\mathsf{InsertResponse}(P, (C_1, \ldots, C_k), f)$ the function which transforms the program P into P' with the response structure $R_f(C_1, \ldots, C_k)$.

Critical code embedding. We assume that, given some program fragment F, it is possible to embed the critical code L into F as follows. The resulting code F' will execute L. If L returns "true", then F' behaves functionally identically to F. Otherwise F' executes some other programmer-specified action. This embedding could be as simple as: *if $L = true$ then run F else quit*. In practice, however, it is desirable to use obfuscation and individualized instances of L, to make it more difficult to circumvent L. We will denote with $\mathsf{CodeEntangle}(L, F)$ the function which returns F' as described above.

4.2 Protection Algorithm

```
Harden(P, L, l, n, k, s, f):
    Flow graph transformation:
        let G = (V, E) ← GraphTransform(P, n)
    Critical code embedding and replication:
        select at random a subset U ⊆ V with |U| = l
        for each v ∈ U do
            v ← CodeEntangle(L, v)
    Check embedding:
        for i = 1 to k do
            select at random v_1, ..., v_s ∈ V
            C_i = InsertCheck(v_1, ..., v_s)
    Creating tampering response:
        InsertResponse(G, (C_1, ..., C_k), f)
```

4.3 Design Goals of Tamper Protection

Before elaborating our model and analysis, we identify desirable properties of a tamper-protection scheme. We believe that our scheme, when instantiated with secure primitives, satisfies those properties. Our model and analysis serve to validate these intuitions.

Tampering response. An obvious goal is to cause improper program operation if tampering is detected. For brevity we will call this a *crash*, but we note that it need not be an actual program crash – it could be slow or unreliable operation, disabling some features, or generally any graceful degradation of program operation.

Thwarting local attacks. The attacker A should gain no information based only on local modifications and observations. For example, changing a single register could result in a crash soon after the change, and A could easily understand how the crash is caused. If tamper-protection is causing it, this provides vital clues to bypassing the protection.

Require multiple failed checks. A secure protection scheme triggers a response only after some minimal number f of checks fails. This makes the task of locating individual checks more difficult.

Hard global analysis. The protection scheme should be embedded as a random structure R in the flow graph G, and discovering R should be necessary for bypassing the protection. A should be able to obtain information about R only through observing walks on G and crashes. Observing the exact memory contents should give no significant advantage to A.

4.4 On Different Security Models

We wish to obtain lower bounds on the complexity of successful attacker A, and quantify them in terms of running time, memory or some other complexity measure. Broadly speaking, there are three approaches:

– *Information-theory*-based approach does not bound the resources of an adversary, but limits the number of probes or input-output queries to a function that is presumed to be a random function.
  Our model is akin to this. If our assumptions hold in practice, then what we derive is a probability bound that an adversary who makes so many probes running the program can learn all the edges he needs to break the system.
– *Complexity-theory* approach considers resource-bounded adversaries, and bases the security of a system on a problem that is intractable. As in cryptography, the instances are picked under some probability distribution.
  The fundamental difficulty in using this approach here is that the program P given to us may be developed by a vast community of programmers, and can only be altered minimally before running into objectionable performance. In particular, the instances do not admit a probabilistic generation model; if we are to base it on graph-theoretic models, no graph problem is known that can be attractive for cryptography, either.
– *Customized constructions* for secure hash and stream ciphers base their security on the best attacks known to the community (DES/AES/SHA-1).

5 Our Model

The model is motivated by practical considerations, namely the available implementations of primitives and currently known attacks and some foreseeable extensions. Denote the protected version of the program by Q, its flow graph by G, the critical code by L, and the attacker by A.

On the one hand, we have A equipped with various tools, including debuggers, data- and control-flow analyzers, graph tools, etc. A is capable of inspecting the code statically and observing its behaviour dynamically. A is also capable of *static tampering* (i.e., changing the code) or *dynamic tampering* (i.e., changing the state of a running program). All of these can be done manually or automated – e.g., using sophisticated programmable debuggers and program-analysis algorithms. A's goal is to ensure that Q runs correctly, even if the critical code L fails.

On the other hand we have the program Q, with critical code L replicated at many locations in Q, and various code-obfuscation transformations applied. If these transformations are secure, then local observations of Q should yield little useful information. Code-obfuscation techniques can make understanding small windows of code difficult. Data-flow obfuscation makes understanding the state of the running program difficult. Flow graph transformations ensure that the flow graph mimics a random graph, with good connectivity between nodes. Fake calls and random calls to replicated code make the execution appear like a random walk on the flow graph. Checks detect tampering, but based on local observations, it is difficult to tell apart checking code from regular code, and check variables from normal ones. Check response is triggered only after sufficiently many checks fail, making the task of pinpointing any check more difficult.

Above we stated some observations and desiderata regarding the protected program and the attacks. We distill them into the following idealized assumptions:

1. Execution of the program induces a random walk on the flow graph at the appropriate level of granularity of clustering the flow graph.
2. Observations restricted to small areas make local variables and code appear to have random values.
3. Tampering local variables or code causes corresponding inserted checks to fail.
4. A sufficient number of failed checks causes the program to crash.
5. In order to prevent the execution of a check, all the locations of variables and code from which the check is initiated need to be identified (e.g., a jump occurs from these locations into the checking code, possibly via computed jumps).

These are indeed simplified idealizations, but we believe that as the implementations of primitives become more secure, or suitable secure hardware is used, the assumptions become closer to reality. Under these assumptions, an attack on a program can be modeled as a game played on the flow graph G of Q.

Informally the game looks as follows. It is played on the flow graph $G = (V, E)$ of Q. A subset U of nodes contains instances of critical code. There are checks $C_1, \ldots, C_k \subseteq V$ in it, each of them consisting of s nodes. Certain subsets of checks are designated as dangerous (the activation structure R contains those dangerous subsets of checks). The attacker A runs Q, and the execution corresponds to a walk on G consisting of independent random steps. In each step A can either tamper with the current node, or leave it alone. A wins the game if he can run the program for at least N steps (N is the parameter), with the following restrictions: (1) Every node that contains the critical code must be tampered; and (2) the program must not crash.

Checks prevent A from winning this game trivially in the following way. Tampering is detected with probability $\mathsf{p}_{\mathsf{detect}}$. A check fails if all its nodes detect tampering. The program crashes when the set of failed checks contains some dangerous set of checks; A should then restart the program; else he cannot win. Finally, the attacker can also try to remove a check: He makes a guess $C = \{v_1, \ldots, v_s\}$, and if $C = C_i$ for some i, then the check C_i becomes effectively disabled – it will no longer fail even if its nodes detect tampering.

Game 1.
BreakingGame($G = (V, E)$, v_0, U, $(C_1, \ldots, C_k)$, R, p, N):

Definitions and terminology. $v_0 \in V$ is called the *entry point*. $U \subseteq V$ is called the *critical code*. $C_i \subseteq V$ ($|C_i| = s$) are called *checks*. R is called the *response structure* and it is a set whose elements are subsets of V (i.e., $x \in R \implies x \subseteq V$). $p \in [0, 1]$ and $N \in \mathbb{N}$ are called the *tamper detection probability*, and the *required running time*, respectively.

Game state. The game state is of the form $(\mathsf{Act}, T, \mathsf{curr}, \mathsf{time}, \mathsf{steps}, \mathsf{crash})$, and its components are called *activated check set*, *tampered node set*, *current node*, *running time*, *step counter* and *crash flag*, respectively.

Game description.

Initial game state. Initial game state is set as follows: $\mathsf{Act} = \{C_1, \ldots, C_k\}$, $T = \emptyset$, $\mathsf{curr} = v_0$, $\mathsf{time} = 0$ and $\mathsf{steps} = 0$.

Game moves. The game proceeds in *steps*, until the player wins or quits. In each step, the player chooses one of the following moves:

RUN. If $\mathsf{curr} \in U$ and $\mathsf{curr} \notin T$ then $\mathsf{crash} \leftarrow 1$. $\mathsf{time} \leftarrow \mathsf{time} + 1$. curr is replaced by its random neighbour in G.

TAMPER. With probability p, curr is added to the set of tampered nodes T. $\mathsf{time} \leftarrow \mathsf{time} + 1$. curr is replaced by its random neighbour in G.

GUESS$(v_1, \ldots, v_s)$. If $\{v_1, \ldots, v_s\} = C_i$ for some i, then C_i is removed from the activated check set Act.

RESET. The game state is reset as follows: $T = \emptyset$, $\mathsf{curr} = v_0$, $\mathsf{time} = 0$ and $\mathsf{crash} = 0$.

QUIT. The game ends. The attacker loses.

Before the next step, the game state is updated as follows. $\mathsf{steps} \leftarrow \mathsf{steps} + 1$. If there is a set of checks $C_{i_1}, \ldots, C_{i_m}$ such that $C_{i_1}, \ldots, C_{i_m} \subseteq T$ and $\{C_{i_1}, \ldots, C_{i_m}\} \cap \mathsf{Act} \in R$, then $\mathsf{crash} \leftarrow 1$. If $\mathsf{time} > N$ and $\mathsf{crash} = 0$ then the attacker wins the game. Otherwise he gets to play the next step.

Note that the sequence of nodes visited between two consecutive RESET moves is a walk on G. Thus, we call the walk that occurs between $(i-1)$-st and i-th reset moves *i-th walk of the game*.

6 Security Analysis

We now consider some statistical attacks on this scheme. To get a more transparent analysis, we will use specific settings of scheme parameters. The analyses indicate that, for these settings, attacks take time exponential in check size s.

Throughout this section we use the following notation. If $W = (v_1, \ldots, v_t)$ is a walk on some graph, we write W^s to denote $\{v_1, \ldots, v_t\}^s$. When no confusion is possible, we write W to denote $\{v_1, \ldots, v_t\}$.

6.1 Case Study: Dense Critical Code, Perfect Checks

1. $U = V$: all nodes of G contain the critical code L,
2. $p = 1$: tampering is detected with certainty,
3. $k = cn$ for some $c > 0$: there are cn checks in the program,
4. threshold $f = cn/2$: half the checks must fail to trigger the response,
5. $N = n^{1+d}$: a succesful attack must run for at least n^{1+d} steps.

Let I_n denote the set of all $((V, E), v_0, V, (C_1, \ldots, C_{cn}), R, 1, n^{1+d})$ where the following hold. (V, E) is an expander graph with $\lambda_2 \leq 1/2$ and $|V| = n$. Every $C_i \subset V$ contains exactly s nodes. $R = \{C_1, \ldots, C_{cn}\}^{cn/2}$, i.e., R contains all $cn/2$-element subsets of $\{C_1, \ldots, C_{cn}\}$. v_0 is arbitrary.

Then by construction, the uniform distribution on I_n is exactly the distribution of BreakingGame instances which correspond to the output of the protection algorithm Harden. We consider two simple attacker's strategies and analyze the expected effort, over random choice of game instance from I_n, and random walks that take place in the game.

Voting attack. This attack is based on the following idea. Let $X = (v_1, \ldots, v_s) \in V^s$ be some choice of s nodes. Suppose A runs the program and tampers it until it crashes, and let $Z = \{v_1, \ldots, v_t\}$ be the corresponding set of tampered nodes. Define $p(X) = \Pr[X \in Z^s]$, the probability that all nodes of X are tampered. We assume, favouring the attacker, that for most check assignments this probability will be concentrated in checks, i.e., there is some δ so that for any check C_i and any non-check $X = (v_1, \ldots, v_s)$, $p(C_i) - p(X) > \delta > 0$. This in particular means that checks are more likely to show up in Z than any other choice of s nodes. A could use this margin to isolate the checks, in the following simple attack.

initialize an s-dimensional $n \times n \times \cdots \times n$ array B to zeros
    (B will store votes for each $(v_1, \ldots, v_s) \in V^s$)
for $i = 1$ to $1/\delta^2$ do
    run A and tamper with it arbitrarily; let W be the set of tampered nodes
    for each $(v_1, \ldots, v_s) \in W$ do
        set $B[v_1, \ldots, v_s] \leftarrow B[v_1, \ldots, v_s] + 1$ *(add one vote for $(v_1, \ldots, v_s)$)*
find the n entries of B with most votes and output their addresses

The complexity of this attack is at least $\Omega(n^s)$. Indeed, barring exponentially unlikely events, each round produces a walk of length at least $n - 2n/s$ (else by Theorem 1 below, too few checks fail for the program to crash). A must now update $(n - 2n/s)^s \approx n^s/e^2$ entries in the array, so this is the least time he needs to spend. Note that this does not even depend on the margin δ. It could be quite big and the above attack would still be too expensive.

Intersection attack. The goal of this attack is to find any single check. A plays m rounds and obtains the walks $W_1, \ldots, W_m$. A hopes that there is at least one check C that fails in each walk $W_1, \ldots, W_m$, and tries to find that check. C

obviously shows up in every W_i; i.e., $C \in B := (W_1 \cap \cdots \cap W_m)^s$. A's search space is thus reduced to B, and he can inspect every candidate from B until he finds C. This strategy, however, takes $n^{\Omega(s)}$ work on average, as indicated below.

By Theorem 1, $1 - 1/(2s) > |W_i|/n > 1 - 2/s$ (with prob. $1 - e^{-O(n)}$). For this simplified analysis, assume that W_i consist of independently drawn samples from V. The expected size of $\cap_{i=1}^{m} W_i$ is lower bounded by $(1-2/s)^m$. Therefore $\mathsf{E}B > n^s(1 - 2/s)^m s \approx n^s e^{-2m}$. Furhtermore $\Pr[C \in W_i]$ by $(|W_i|/n)^s$, so $\Pr[C \in \cap_{i=1}^{m} W_i] \leq (|W_i|/n)^{sm} \leq (1 - 1/(2s))^{sm} \approx e^{-m/2}$. Therefore, to get success probability ϵ, the attacker must set $m < 2 \log \epsilon$. But then his work (i.e., the size of B) is at least $n^s e^{-2m} > n^s \epsilon^4$.

6.2 Other Parameter Settings and Models

A more practical setting is the one where $U \subset V$ and $0 < p < 1$; i.e., critical code is distributed only through a fraction of the program code, and checks have a chance to miss tampering. In this case, the algorithm Harden should be modified to insert checks only in U. Using some random walk lemmas (see for example [9]) and techniques similar to those of the previous section, one can prove similar lower bounds. We do not provide details in this article.

In a more realistic model, the attack game can be changed to allow A to choose some steps in the walk, instead of just passively observing them. One could for example let A choose every other step adversarially. If A's strategy is non-adaptive (i.e., each adversarial step depends only on the current node), then the results of [8] can be used to analyze attack complexity, and derive bounds similar to those of the previous section. We do not provide any details in this article.

7 Conclusion and Future Work

This article presented a new graph-based framework for modeling and implementing specific tamper-resistance algorithms. Our scheme may yield practical program-transformation tools to harden software against malicious patching and interference. A crucial improvement over today's "ad hoc" protection methods is an attack-resistance model, which can help estimate how long particular protected applications will remain unbroken in practice. This is important in various business scenarios (e.g., DRM and software licensing), where measurable attack resistance can prevent unexpected breaches and enforce a consistent revenue stream.

Future work will involve making our models and assumptions more realistic, as well as performing more implementation and experimental verification. Upcoming efforts will study the exact theoretical resistance offered by our algorithms, and also develop new algorithms along similar lines. We note that theoretical impossibility results on obfuscation [3, 11] do not pose roadblocks to development of such algorithms, because the attack resistance we require need not be exponential or even superpolynomial. As long as our techniques

can predict such resistance (or lack thereof) accurately for typical programs, our approach should be useful in practice.

A Graph Lemmas

Lemma 1. *Consider a walk W which tampers $t = n - n/2s$ distinct nodes. Then:*

1. *Expected number of failed checks is $cn/\sqrt{e} + \epsilon(s)$ for some $\epsilon(s) \in o(s)$.*
2. *For $1 - e^{-O(n)}$ fraction of check assignments, the number of failed checks is at least $0.5cn$.*

Proof. Let $T \subseteq V$ be the set of tampered nodes. Denote $\mu = |T|/|V| = t/n$. Let p be the probability that a randomly chosen check $C = (v_1, \ldots, v_s)$ is contained in T. For sufficiently large s, we have $p = \mu^s = (1 - 1/2s)^s \approx 1/\sqrt{e}$.

1. There are cn checks, so the expected number of checks contained in T is cnp and this converges to $cn/\sqrt{e} > 0.6cn$ as $n \to \infty$.
2. Applying a Chernoff bound, one gets that the number of failed checks is exponentially unlikely to fall below $0.5cn$.

Lemma 2. *Consider a walk W which tampers at most $t = n - 2n/s$ distinct nodes. For $1 - e^{-O(n)}$ fraction of check assignments, the number of failed checks is at most $cn/4$.*

Proof. Let T, μ and p be as in the proof of Lemma 1. Then $p = \mu^s = (1-2/s)^s = ((1-2/s)^{(s/2)})^2$. For sufficiently large s we have $p \approx 1/e^2$. The expected number of failed checks is thus cn/e^2, and using a Chernoff bound one gets that at most $cn/4$ checks fail, except with probability $1 - e^{-O(n)}$.

It is easy to do a "quantifier switch" to make the probabilities of Lemmas 1,2 over random walks, using the following simple lemma.

Lemma 3 (Pigeonhole principle variant). *Let $I(a, b)$ ($a \in A$, $b \in B$) be a 0-1 matrix. Let $p_b = \Pr_{a \in A}[I(a, b) = 1]$ and let $p = \Pr_{(a,b) \in A \times B}[I(a, b) = 1]$. If $p \leq \epsilon$ then*

$$\Pr_{b \in B}[p_b \geq \sqrt{\epsilon}] \leq \sqrt{\epsilon}.$$

Using Lemma 3, Lemmas 1,2 and taking into account that $U = V$, it is easy to show:

Theorem 1. *For $1 - e^{-O(n)}$ fraction of check assignments the following hold:*

1. *$1 - e^{-O(n)}$ fraction of walks shorter than $n - 2n/s$ do not crash.*
2. *for any constant $d > 0$, $1 - e^{-O(n)}$ fraction of walks longer than n^{1+d} crash.*

Proof. The first claim follows directly from Lemma 2. For the second claim, note that a random walk of length n^{1+d} covers G with probability $1 - e^{-O(n)}$. Therefore the subset U of nodes containing the critical code is covered. To avoid crashing due to untampered execution of critical code, the attacker must tamper with every node on the walk. So more than $n - n/2s$ nodes are tampered, and by Lemma 1 the walk with probability $1 - e^{-O(n)}$.

References

1. Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: Virtualization for diversified tamper-resistance. In K. Kurosawa, R. Safavi-Naini, and M. Yung, editors, *Proceedings of the Sixth ACM Workshop on Digital Rights Management*, pages 47–57, Washington, 10 2006. ACM.
2. David Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, London, UK, 1996. Springer-Verlag.
3. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Advances in cryptology - CRYPTO '01, Lecture Notes in Computer Science*, 2139:1–18, 2001. http://www.wisdom.weizmann.ac.il/ boaz/Papers/obfuscate.ps.
4. Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *IH '02: 5th International Workshop on Information Hiding*, pages 400–414, London, UK, 2003. Springer-Verlag.
5. Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.
6. Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL'98*, pages 184–196, 1998.
7. DataRescue. IDA Pro.
8. Murali K. Ganapathy. Robust mixing. In Josep Díaz, Klaus Jansen, José D. P. Rolim, and Uri Zwick, editors, *APPROX-RANDOM*, volume 4110 of *Lecture Notes in Computer Science*, pages 351–362. Springer, 2006.
9. David Gillman. A chernoff bound for random walks on expander graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 680–691, 1993.
10. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
11. Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 553–562, Washington, DC, USA, 2005. IEEE Computer Society.
12. B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *In Eurocrypt '04*, 2004.
13. Adi Shamir and Nicko van Someren. Playing "hide and seek" with stored keys. *Lecture Notes in Computer Science*, 1648:118–124, 1999.
14. Gang Tan, Yuqun Chen, and Mariusz H. Jakubowski. Delayed and controlled failures in tamper-resistant systems. In *Proceedings of 8th Information Hiding Workshop*, 2006.
15. Paul C. van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Trans. Dependable Secur. Comput.*, 2(2):82–92, 2005.
16. Avinash Varadarajan and Ramarathnam Venkatesan. Limited obliviousness for data structures and efficient execution of programs. Unpublished manuscript.
17. Zheng Wang, Ken Pierce, and Scott McFarling. Bmat - a binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000.
18. Hoeteck Wee. On obfuscating point functions. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532, New York, NY, USA, 2005. ACM Press.