Oblivious Hashing: A Stealthy Software Integrity Verification Primitive

Yuqun Chen[1], Ramarathnam Venkatesan[1], Matthew Cary[2], Ruoming Pang[3], Saurabh Sinha[2], and Mariusz H. Jakubowski[1]

[1]Microsoft Research, One Microsoft Way, Redmond, WA 98052
[2] University of Washington, Box 352350, Seattle, WA 98195
[3] Princeton University, 35 Olden Street, Princeton, NJ 08544

{yuqunc, venkie, mariuszj}@microsoft.com
{mcary, saurabh}@cs.washington.edu
rpang@cs.princeton.edu

Abstract. We describe a novel software verification primitive called Oblivious Hashing. Unlike previous techniques that mainly verify the static shape of code, this primitive allows implicit computation of a hash value based on the actual execution (i.e., space-time history of computation) of the code. We also discuss its applications in local software tamper resistance and remote code authentication.

1. Introduction

A major challenge in Software Tamper Resistance is finding a stealthy and robust primitive to ascertain the operational correctness of protected software. A prevalent method used today verifies the shape of the code, and sometimes critical data, before or during the runtime. It is accomplished by computing a cryptographic checksum on one or more segments of the code that is being protected.

The shape-verification approach has many drawbacks of which we mention two here. First, it is quite straightforward to detect the verification routine, owing to the atypical nature of the operation, since most applications do not read their own code segments. Second, this approach fails to detect certain behavioral or data modifications to the software. For example, a hacker can temporarily change the return result by patching a register, right before the protected function returns, without having to alter the code.

To address such problems, we propose a new concept called Oblivious Hashing (**OH**). The main idea is to hash the execution trace of a piece of code, thereby allowing us to verify probabilistically or deterministically the run-time behavior of the software. We accomplish this by injecting additional computation (*hashing code*) into the software (*host code*). The hashing code implicitly computes a hash value from the dynamic execution context of the host code. A main feature of our injection method is to blend the hashing code seamlessly with the host code, making them locally indistinguishable and thus difficult to separate without non-trivial effort to run and observe the program's execution repeatedly.

Given **OH** as a primitive, a number of techniques can be used to make it an effective tool for tamper resistance. At the minimum, there must be stealthy ways for checking and acting upon the execution hashes; invocations of the checks must be networked into a graph so as to make it combinatorially hard to identify the underlying graph structure and remove these checks [1,11]. These techniques are critical to tamper resistance regardless of the software verification mechanism used. In this paper we focus on the oblivious hashing primitive itself, describing its concept and some implementation details. We also discuss its application to local tamper resistance and remote code authentication.

2. Related Work

Tamper-resistance techniques traditionally took advantage of software- and hardware-specific features that were often un-documented (e.g., hidden functionality and "reserved" processor instructions). Unfortunately, special features are invariably limited in quantity and unchangeable over time. Typically, it did not take long before an unrelenting hacker uncovered the "trick" and completely defeated the protection. Not surprisingly, the security-by-obscurity approach has gradually given ground to more systematic approaches.

The last few years have seen active development of computation-based techniques in software tamper resistance. A typical such system incorporates both software integrity verification and software obfuscation as its two main weapons against tampering.

Among software integrity verification methods, computing a checksum (or hash) of code bytes (or, Code Checksum) is perhaps the oldest. An obvious direct derivative of the long studied areas of hashing for searching and sorting [18] and that of message authentication codes in cryptography [19], it is straightforward to implement and can be made quite efficient [1, 14, 15]. The main drawback of code checksum is that reading the code segment tends to stick out as an atypical operation during execution. The hackers can usually pin-point the checks by setting breakpoints or examining the code. Spreading many smaller checks over time and space, repeating atypical operations all over the code are the obvious heuristics to mitigate this problem. Furthermore, since this method only verifies the static shape of the code, it cannot detect certain run-time attacks whereby the hacker patches the instructions or return value (often held in a register) temporarily.

Software obfuscation [1, 3, 4, 5] is another general but complementary technique that tries to make it difficult for attackers to understand and modify code in a useful manner. Most commercial obfuscation deployments involve proprietary technologies without many published implementation details, except on sites run for and by hackers who enjoy reverse engineering. A popular technique in software obfuscation is Code Encryption. Unless almost all the code is encrypted, use of this technique is easy to identify and attack. It does, however, help prevent straightforward disassembly and patching [1]. In the extreme case, code can even be decrypted one instruction at a time and custom-generated on the fly. On processor architectures with variable-length instructions, for example Intel x86, one may also devise some anti-disassembly and anti-decompilation techniques by embedding instructions within other instructions. Furthermore, since a debugger is a primary tool of crackers, debugger-detection and -disabling techniques may be viewed as useful deterrents. Some protection methodologies have strived to detect and disable general tools that might be used for observation and modification of running programs, such as described in [9, 10, 20s].

Historically, hardware and software debugging methods use printing out the trace of an execution. Our contribution can be viewed as a (probabilistic) compression of the trace from a security point of view. This is quite sensitive to the attack model, which in our case is that the program runs in a system owned by the adversary, who can use program analysis tools to detect, thwart and undo the checks and protection. This has to be contrasted with protection from downloaded code (where the adversary is external and the sandboxing can be an appropriate measure) or running a given code in an untrusted remote system and checking if the computations are properly carried out. The latter problem in a distributed execution environment is studied in [16], where a suitably embedded trace-gathering code ensures that the remote computer sends the result and the trace which can be locally cross-checked. Our trace-collection differs fundamentally on how it is done, used and the goals themselves. Our emphasis is on stealth and security via adequate randomization of the end results**. OH** gathers only a small subset of the execution states and state reduces the sequence of state changes to a small hash value to approximate and implement a light-weight progressive, one-way hash function designed to fingerprint the computation. It uses random keys and stresses the production of hash values that are hard to guess even if one has a functionally equivalent program that is not the exact copy of it. It is suitable for both local tamper resistance and remote execution verification.

An orthogonal but seemingly related task is program (or result) checking [2, 7, 8, 13] that attempts to verify the input-output behavior of a program, but not its full execution behavior as **OH** does. These techniques are applicable only to problems on algebraic domains (whereas **OH** is intended for general software) and have no stealth criterion. Ideally one would have an algorithm that transform any given program into an “obfuscated” version which, roughly speaking, can be executed in a black-box fashion as if on a secure co-processor. But this is impossible or unlikely [17] in many models, because there exist programs that do not admit such an algorithmic transformation. However, for large, practical programs (e.g., not the “hello world” type), under suitable engineering assumptions one still may be able to derive quantifiably secure systems, where a primitive such as oblivious hashing can play a significant role.

3. Oblivious Hashing

We first present the **OH** technique using an abstract, and then describe a software-only implementation approach.

3.1 Abstract Model

In a simplified model of computation, a function (or a program)[1] is represented by a sequence of abstract machine instructions **I**=$\{i_1, i_2, \ldots, i_N\}$ that read and write memory locations **M**=$\{m_1, m_2, \ldots, m_K\}$, the initial configuration of the memory $\mathbf{M}^0$, an instruction counter **C** and its initial value $\mathbf{C}^0$. Our main idea is to capture the function's execution trace **T** from which to compute a hash value **H**, as shown in Figure 1. Since the trace reflects the actual execution, the hash value thus computed serves as a robust signature on the function's behavior. This value is a function of the code, data, the initial configuration of the machine, and the input parameter **P**:

$$H \leftarrow H\ (T) \leftarrow H\ (I, M, C^0, M^0, P) \qquad \{ \leftarrow \text{means “depends on”} \}$$

We observe that in this model, external environment to the function is encoded in the initial memory configuration M^0. A slight modification of the code and/or data is likely to cause the hash value to change, provided that T contains sufficient information from the actual execution.

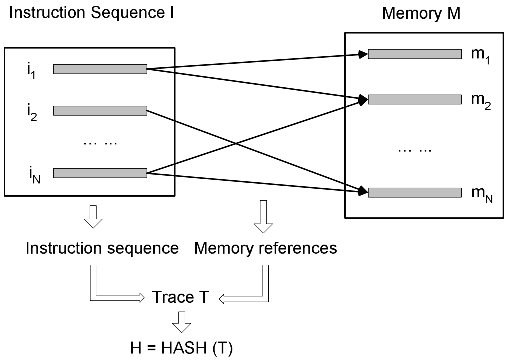

Figure 1: an abstract model for oblivious hashing

Attack model: The attacker changes the instruction sequence and memory content during runtime in order to produce a correct hash value for a given set of inputs (which can be exorbitantly large).

Note that if we apply OH only to a portion of the code, e.g., licensing verification, the attacker can bypass the licensing check by modifying the rest of the application. In

[1] Unless explicitly stated, the terms “function” and “program” are considered inter-changeable.

general, protection must be extensive and spread out throughout the application. At the minimum, all the functions on the calling-hierarchy to the protected code must also be protected by OH and other mechanisms.

The ideal trace T should include memory references made by each instruction and the instruction itself. One way to accomplish this is by using special-purpose hardware built into the microprocessor. A much less expensive and more flexible approach is to implement the hashing model in software.

3.2 The Software Approach

The naïve software approach is to build a machine simulator that mimics the behavior of the hashing co-processor. But this is quite inefficient and vulnerable to a total break by a one-time attack on the simulator. A practical and efficient implementation can be done via code injection, which is common practice employed by profilers and bounds checkers. We inject into the host code "monitor" instructions that capture each step of the computation and compute the oblivious-hashing value as the computation proceeds, as illustrated in Figure 2. The hashing instructions, colored black in the figure, take the results of previous instructions and apply them to the hash values stored in main memory. The diagram illustrates multiple memory locations that jointly store the intermediate and final result. Note that these hashing instructions are the same kind of instructions as in the original software; they read and write data locations just like other instructions. With proper care when injecting the hashing instructions, we can make them blend seamlessly into the to-be-verified software both in appearance and during execution.

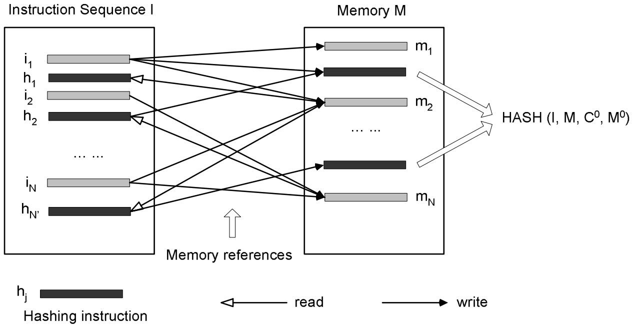

Figure 2: Software-only Oblivious Hashing

Compared with the hardware approach, the code-injection approach can achieve similar degree of efficiency, and yet requires no modifications to the underlying hardware and operating system.

We term our method of hashing computation "oblivious hashing," as an attacker is oblivious to the fact that part of the software is computing an execution hash value during the normal computation.

4. Implementation

To implement **OH**, we preferred not to perform binary editing on the compiled code and chose to work with a higher-level representation. Besides the obvious high overhead, binary-level code injection suffers from several practical constraints that make it difficult to obfuscate the identity of hashing instructions. For example, due to the register-oriented nature of modern processor architecture, our hashing injections have to use registers to store intermediate values. However, register allocation is already fixed by the time object code is generated and it is inefficient to reverse-engineer the register-allocation at the binary level, and find free registers for use by hashing code. Pushing-and-popping used registers will make the hashing instructions obvious and thus allow them to be easily spotted. Hence, injection of the hashing instructions at a higher level is preferred.

4.1 Syntax-tree Modification

This higher level that we have in mind is the syntax tree produced in the parsing stage of the compilation process. The syntax tree contains explicit dependency information, which makes it straightforward to insert additional statements while preserving the correctness of original computation.

The second advantage in inserting OH code in the syntax tree is speed. The syntax-tree representation can be thought of as a higher-level abstract machine: Instead of instructions, we have expressions and statements. Instrumenting the syntax tree is more efficient than instrumenting the machine code, because many intermediate computations are saved from hashing. For example, to hash the C statement **X=Y+A*B–C/D**, we need only hash the final assignment to **X**, because the intermediate value computations do not cause changes to the state of the function (or program). Were the attacker to change the intermediate computation, he would still be caught if **X** is assigned an incorrect value.

The third advantage is that the optimizations performed in the subsequent stage may interleave the hashing code with the host code, making detection of the hashing code more difficult. Temporary variables used by the hashing code are allocated identically to host-code variables by the second phase of the compiler, and thus are not easily distinguishable from host-code variables.

Finally, instrumenting the syntax tree makes oblivious hashing machine-independent. Being used in early stages of compilation, syntax trees contain little or none information on the machine architecture. Oblivious hashing will thus produce the same results regardless of the final machine platform. This is especially useful for platforms such as MSIL and Java.

We implemented OH for the Microsoft Visual C/C++ compiler, version 7.0. The compiler emits intermediate-level representation, C Intermediate Language (CIL) for every source file. The CIL representation is just like any compiler intermediate representation. The CIL files are stored temporarily on disk before being passed on to the compiler backend. The CIL representation contains just enough information for the C-style backend to perform optimization and code generation. Though information on complex types is lost in CIL, we found the representation adequate for adding OH code[3].

4.2 Hashing Sites

Our next step is to determine the kind of programming constructs to hash. Assignments and control flows are two natural choices. Assignments change the state of a function or a program, and hence are the largest indicator of a program's behavior. We found, however, that many functions served only to make high-level control flow decisions, and that few assignments were performed in coming to these decisions. Adding control flow expressions to our set of hash sites made the final hash value much more sensitive to program behavior.

While a more complete system could hash things such as function arguments, hashing assignments and control flows captures most of the dynamic behavior of a program . For example, since most function arguments are simply variables whose value has been computed in a previous assignment, hashing function arguments would not add much extra information to the hash.

Assignment and expressions: The C operator we rely on is the comma operator, which is rarely used in programming. Note that the comma operator is not the same as the comma used to separate function arguments. A C expression (exp_1, exp_2, ..., exp_k) will evaluate exp_1 through exp_k, and have the value of exp_k. Thus, an assignment is transformed in the following way:

a = exp   is transformed to   a = (t = exp, HASH(t), t)

Conditional expressions are similarly transformed:

if (exp) {…}   is transformed to   if ((t=exp, HASH(t), t)) {…}

Using the comma operator and a temporary variable *t*, this transformation lets us intercept the value of assignment, and at the same time preserve the C semantics that the assignment operator has a value of the assigned value. Here expression HASH(t) is an inline macro, as an explicit function call would violate our principle of making the hash code difficult to detect and not centrally located.

Control flows: To capture the control flow within a function, we inject one or more hashing instructions within each basic block. In our syntax-tree representation of the

[3] We later found out that CIL is inadequate for using static analysis to identify “unhashable” statements. A re-writable syntax tree with richer information would have been more suitable. Unfortunately, such a tool does not exist on Windows platform yet.

function, a basic block is identified by a label that is either specified by the programmer or generated by the compiler front end. In the CIL representation, it is quite straightforward to locate the labels and inject hashing operations between two adjacent ones.

Our OH-injection tool also allows the user to specify the percentage of a function to be hashed. This gives the user to control the amount of overhead, in terms of both code size increase and run-time overhead. This feature is especially useful when applying OH in remote code authentication (described in Section 5.2).

4.3 Hash Computation

The hashing instructions are executed every time the host code runs, and thus they must comprise inexpensive operations. Fortunately, our situation is different from message authentication codes (MAC) so that such performance optimizations are possible. A MAC is applied to a piece of static data each time, whereas an oblivious hash is derived from the execution of the host code or a fraction of it. The execution of the host code depends on the input parameters. The attacker may be able to carefully modify the code so that the tampered code produces the correct oblivious hash for a given input parameter. But it would be much more difficult to do this for many different inputs. In fact, if a code segment yields the same **OH** values for sufficiently large number of inputs, there may be only one such segment. An analogy to linear algebra will help illustrate this point.

> Let us imagine that the code including the hashing instructions is a vector of numbers denoted as Ω, that each input is a test vector, and that the dot product between the code vector and the input vector yields an oblivious hash. A collection of inputs I_1, I_2, .., I_n forms a test matrix Γ such that $\Gamma \cdot \Omega = H = [H_1, H_2, \ldots, H_n]$, where H_i is the corresponding hash value. Given a sufficient number of orthogonal vectors in Γ, Ω is uniquely determined.

The above analogy is a gross over-simplification of oblivious hashing: The instructions in the hashed code do not correspond to numbers, and the inter-relationship between instructions is far more complicated than that in a set of numbers. Even a simple analogy like this shows the difficulty faced by the hacker: It is very difficult to tamper with the code (i.e., the vector Ω) while producing correct hash results (i.e., preserving the relationship $\Gamma \cdot \Omega = H$).

In reality, the amount of binary modifications that the hackers can do is quite limited, due to the lack of sufficient information in a compiled, released binary executable. The common operations employed by the hackers are replacing existing instructions with different ones, setting debug breakpoints to hijack controls from the code, and intercepting API calls to the underlying operating system. Given that the patched instructions must not result in program crashing, the attacker can generally patch only a tiny number of instructions. This greatly restricts his freedom in altering the code and yet producing the same hash results for more than one challenge input. We conjecture that weaker but efficient operations are adequate to compute the oblivious hash.

In one of our implementations, the hash result is kept in a **logical** array of 32-bit words, initialized to a random pattern during compile time. Each hashing operation **HASH(val)** takes the general form

$$\mathbf{H[i] = H[i]\ op_1\ H[j]\ op_2\ X\ \{\ op_3\ random\_constant\ \}}$$

where **H[i]** and **H[j]** are two elements in the hash result array, randomly chosen for the hashing operation. **X** is a random constant generated at the compile time. The three operators, op_1, $\mathbf{op_2}$, and $\mathbf{op_3}$, are chosen randomly from a set of arithmetic and bit operators at the compile time. The operation "$\mathbf{op_3}$ **random_constant"** in the curly braces are optional. We can also profile the original program to bias the operation selection so as to avoid creating "obvious" hashing code such as one containing too many XOR operations.

In principle, the hash array need not be contiguously allocated; it can be made up of a number of unrelated 32-bit variables scattered around in the heap. For convenience, our current implementation allocates the hash array contiguously, though minor change in our software will easily remove this artifact.

The hash storage need not be in the global variables, either. Our hash injection tool can also modify the hashed functions[4] and their call sites to have the hash variables passed by pointer. For example, in order to verify a function F(x, y) using oblivious hashing, the caller can simply allocate a hash variable h on its stack and passes it to F as an additional argument, e.g., F(x, y, &h). Note that our injection software can place the hash variable at an arbitrary place on hashed function's argument list, as long as its position is consistent for all call (and hashing) sites.

Code Sample: In the appendix, we illustrate oblivious hashing using a factorial function written in C. We compile the program using the Microsoft C/C++ compiler with the "-O2 –GB" optimization flags. Disassembly listing of the original function and two hashed versions are provided. In the first hashed version, 50% of the hashable statements are randomly chosen for hashing. In the second version, OH is performed on the entire function. For both versions, the 32-bit OH result is passed back and forth via an additional pointer argument to the function, which is created automatically by our OH-injection tool.

[4] We here address only C functions, and omit decorated names for modified C++ functions.

4.4 Unhashable Statements

So far we have assumed a simplistic function that is deterministic with respect to input data[5]. In reality, some code depends on external environment such as time of the day, user identity, etc., that is hard to control. The internal memory allocator may return different addresses depending on how the thread is scheduled. Unpredictable data, such as external input or return values of system calls, can cause oblivious hashes to vary arbitrarily. As a result, only a portion of the host code executes deterministically with respect to the input parameters. The question of what to fingerprint for an execution is a major problem.

There are two obvious approaches to solving this problem. One can apply the usual static-analysis techniques to conservatively locate the deterministic portion of the host code. Alternatively, one can gather run-time traces on some test inputs, and use the traces to determine the portion of the execution path that remains the same across the test inputs. Once these “unhashable” expressions and side effects are determined, we can exclude them from the hash computation. We may also determine and specify “unhashable” sites manually, though automated analysis would prove much more convenient and less prone to human errors.

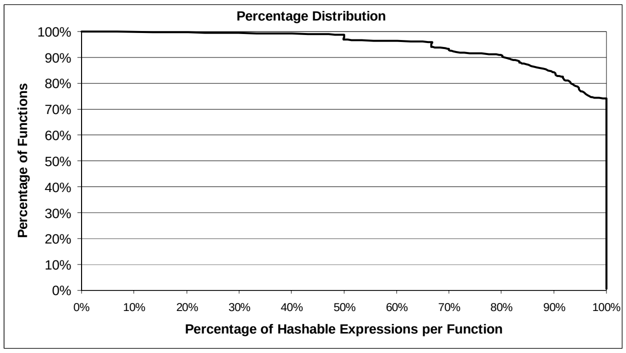

Figure 3: Hashable expressions per function

We experimented with the second approach on a commercial application with anti-tampering features. We instrumented the program to produce a trace of expression values that we are interested in. We then ran the instrumented program multiple times, in all interesting execution contexts, and post-processed the tracing output to determine which expressions were constant across runs. The graph above (`Figure 3`) shows the distribution of functions according to the amount of hashable expressions in each. The results are quite telling. We found that even using such a strong requirement still left enough hashable expressions to provide good code coverage for our test program. However, we do want to point out that our test inputs may not

[5] Accesses to global data inside the function can be parameterized with additional pass-by-reference parameters.

exhaust all possible code paths, although we have reasons to believe that they do by visual inspection. In production code the situation may well be different. A conservative approach based on static analysis would be more assuring. Static analysis for oblivious hashing is still an on-going work.

5. Applications of Oblivious Hashing

As a method to verify the behavior of programs, oblivious hashing helps solve some currently important problems in both local-software security and network-oriented computing. For software running locally on a single machine, oblivious hashing can prevent tampering by either a malicious hacker or malware (viruses, worms, and Trojans). For client-server applications, oblivious hashing can prove a client's authenticity and proper functioning to a server, and vice versa.

5.1 Local Software Tamper Resistance

For verifying the proper operation of a program in a simple manner, oblivious hashes can be used much like typical hashes or checksums of code bytes: The program computes hashes and compares them against pre-stored values. However, if done naïvely (e.g., with one or a few simple Boolean checks), this is not likely to be strong; the problem of easily patched Boolean comparisons is the same as with verification of code-byte checksums. Additionally, oblivious hashes introduce a number of unique issues:

> **Pre-computation of correct hashes**: Unlike a code-byte hash, an oblivious hash is "active" in that the code to be protected must run (or be simulated) in order for the hash to be produced.
>
> **Security coverage over code paths**: An oblivious hash depends on the exact path through a program, as determined by input data. If execution does not reach some part of a program during hash computation, that part is not hashed and thus unprotected.
>
> **Unhashable data**: As described earlier, data that are too variable or not predictable typically cannot be hashed obliviously.

One strategy for addressing the above issues is as follows. Prior to shipping a protected application, we choose a number of specific functions and inputs that exercise security-sensitive code paths. We then run the given functions on the specified inputs, saving the resulting oblivious hashes. Our security requirements and knowledge of the application's semantics determine the functions and inputs used for oblivious hashing. Alternately, static analysis or hints from the developer could automate the process of determining what to hash.

Another unique issue with oblivious hashing involves cross-checking of different code sections, a technique essential for increasing the complexity of breaking protection [11]. With code-byte hashes, two code segments, A and B, can cross-check each other simply by verifying each other's hashes. With oblivious hashing, however, a cross-check involves mutually recursive calls. Without special methods of termination, this recursion is infinite. Unfortunately, explicit termination measures could point the hacker's attention to oblivious-hashing code, as well as create significant complexity when more involved cross-checking graphs and indirect recursion are present. A promising approach is to invoke cross-checks probabilistically. It's a topic of our current research on **OH**.

5.2 Remote Code Authentication

Autonomy is the trend in network services: Software client agents act on behalf the users; servers talk to one another without human intervention. There is a growing demand for robust authentication of remote software entities.

Remote code authentication is the problem of verifying the identity of a remote program. Authentication can occur at all levels: between a client and a server, between two servers, and between two communicating clients. Unless specifically noted, the terms "client" and "server" refer to the application software that runs on each client or server machine.

Remote code authentication differs from the usual user-based authentication, in that it strives to verify the identity (or the authenticity) of the communicating application as opposed to the user who is running the software. Unlike passwords, a piece of code is an active entity. Verifying its dynamic behavior yields stronger authentication than checking its static shape, as the hacker can easily fool the latter mechanism by keeping a copy of the original, unhacked code around.

Oblivious Hashing is an ideal fit for remote code authentication, as the oblivious hash represents a fingerprint of the code execution. In a simple embodiment (Figure 4), machine A authenticates a program P on machine B by generating a random input X to P and sending it to B. An identical copy of P (or a critical portion of P) is also present on machine A. Both A and B run P on the random input X and derive an oblivious hash value H. To forestall the man-in-the-middle attack, the hash value H is not transmitted over the wire. Instead, a session encryption key is derived from H; it is used to encrypt all subsequent communication between A and B. If B and A run the same version of P, they ought to derive the same session key. This is how A can verify the authenticity of program P on a remote machine B.

An adversary can still attempt to crack this system by extracting the hashed computation from P or calling the computation directly. The problem thus becomes reverse-engineering the program P, which can be made quite difficult by usual obfuscation techniques such as OH for local tamper resistance, code encryption and anti-debugging mechanisms.

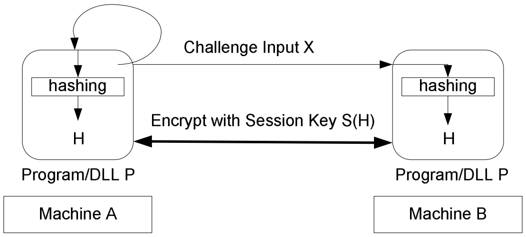

Figure 4: Oblivious Hashing used in remote code authentication.

6. Conclusions

In this paper we presented a novel software integrity verification primitive, Oblivious Hashing**,** which implicitly computes a fingerprint of a code fragment based on its actual execution. Its construction makes it possible to thwart attacks using automatic program analysis tools or other static methods. This new method verifies the intended behavior of a piece of code by running it and obtaining the resulting fingerprint. It is more implicit than the traditional shape-verification methods, and addresses some attacks that cannot be detected by the latter.

We also presented a software-only implementation that injects and blends hashing code into the syntax-tree representation of the program. This approach avoids many problems faced by a binary-editing method. Only critical expressions are hashed by our method. For example, intermediate computations are skipped as they do not directly affect the program state. The result is much lower overhead. Manipulating at the syntax-tree level also allows the injected code to blend well with the host code, achieving high degree of secrecy. This technique is suitable for local software tamper resistance and remote code authentication.

Our experiment with a real-world program showed that a large portion of the code can be obliviously hashed for a majority of functions, which is encouraging. Our method also raises questions about how to adapt existing program analysis techniques to suit our purposes.

Oblivious Hashing has its own limitations. For example, it cannot detect x86 debugger breakpoints implemented by using the **INT3** instruction, whereas the traditional code-checksum method can. A robust software tamper-resistance solution should incorporate both OH and other methods.

7. Acknowledgement

We thank the reviewers for their feedback and pointers to related literature.

References

1. D. Aucsmith, “Tamper Resistant Software: An Implementation,” in Proceedings of the First International Workshop on Information Hiding , May 1996.
2. M. Blum and S. Kannan, "Designing Programs That Check Their Work,” in Proceedings of ACM Symposium on Theory of Computing, pgs 86-97, 1989.
3. C. Collberg, C. Thomborson and D. Low, “Breaking Abstractions and Unstructuring Data Structures,” in Proceedings of IEEE International Conference on Computer Languages, ICCL'98, May 1998.
4. C. Collberg, C. Thomborson and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", in Proceedings of Symposium on Principles of Programming Languages, pp. 184-196, 1998.
5. C. Collberg and C. Thomborson, “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.”
6. Cloakware Corporation, http://www.cloakware.com.
7. F. Ergun, S. Kannan, S. R. Kumar, R. Rubinfeld and M. Viswanathan, “Spot-Checkers,” in Proceedings of ACM Symposium on Theory of Computing, pgs 259 – 268, 1998.
8. F. Ergun, S. R. Kumar and D. Sivakumar, “Self-Testing Without the Generator Bottleneck,” SIAM Journal of Computing, vol. 29, no. 5, pgs 1630—1651, 2000.
9. G. Hunt and D. Brubacher, “Detours: Binary Interception of Win32 Functions,” in Proceedings of the 3rd USENIX Windows NT Symposium, pgs 135 – 143, July 1999.
10. Sysinternals, http://www.sysinternals.com.
11. R. Venkatesan, V. Vazirani, and S. Sinha, “A Graph Theoretic Approach to Software Watermarking,” in Proceedings of the Fourth International Workshop on Information Hiding, April 2001.
12. C. Wang, J. Hill, J. Knight and J. Davidson, “Software Tamper Resistance: Obstructing Static Analysis of Programs,” Technical Report CS-2000-12, University of Virginia, December 2000.
13. H. Wasserman and M. Blum, “Software Reliability via Run-Time Result-Checking,” Journal of ACM, vol. 44, no. 6, pgs 826 – 849, 1997.
14. B. Horne, L. Matheson, C. Sheehan and R. Tarjan, “Dynamic Self-Checking Techniques for Improved Tamper Resistance,” in Proceedings of the Workshop on Security and Privacy in Digital Rights Management, November 2001.
15. H. Chang and M. Atallah, “Protecting Software Code by Guards,” in Proceedings of the Workshop on Security and Privacy in Digital Rights Management, November 2001.
16. F. Monrose, P. Wyckoff, and A. Rubin, “Distributed Execution with Remote Audit,” in Proceedings of the ISOC Network and Distributed System Security (NDSS) Symposium, February 1999.
17. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.Vadhan and K. Yang, “On the (impossibility) of Obfuscating Programs,” *Advances in Cryptology - CRYPTO `01,* vol. 2139 of Springer-Verlag Lecture Notes in Computer Science, pp. 1-18, August 19-23, 2001.
18. D. Knuth, “The Art of Computer Programming, Volume 2, Seminumerical Algorithms,” Addison-Wesley Publishing Company, Inc., 1973.
19. A. Menezes, P. van Oorschot and S. Vanstone, “Handbook of Applied Cryptography,” CRC Press, 1997.
20. SoftICE debugger, Compuware Corporation, http://www.compuware.com.

Appendix

C source code

```
unsigned int factorial(int n)
{
     unsigned int fact;
     for (fact=1; n>0; n--) fact=fact*n;
     return fact;
}
```

Assembly list of the original, unhashed function

```
_factorial:
  00000000: mov         ecx,dword ptr [esp+4]
  00000004: test        ecx,ecx
  00000006: mov         eax,1
  0000000B: jle         00000018
  0000000D: lea         ecx,[ecx]
  00000010: imul        eax,ecx
  00000013: dec         ecx
  00000014: test        ecx,ecx
  00000016: jg          00000010
  00000018: ret
```

Assembly listing of the 50%-hashed function

```
_factorial:
  00000000: mov         ecx,dword ptr [esp+4]
  00000004: test        ecx,ecx
  00000006: mov         eax,1
  0000000B: jle         00000026
  0000000D: push        esi
  0000000E: mov         esi,dword ptr [esp+0Ch]
  00000012: imul        eax,ecx
  00000015: mov         edx,ecx
  00000017: dec         ecx
  00000018: xor         esi,edx
  0000001A: test        ecx,ecx
  0000001C: jg          00000012
  0000001E: mov         eax,dword ptr [esp+0Ch]
  00000022: mov         dword ptr [eax],esi
  00000024: pop         esi
  00000025: ret
  00000026: mov         ecx,dword ptr [esp+8]
  0000002A: mov         edx,dword ptr [esp+8]
  0000002E: mov         dword ptr [ecx],edx
  00000030: ret
```

Assembly listing of the 100%-hashed function

```
_factorial:
  00000000: mov         ecx,dword ptr [esp+4]
```

```
00000004: xor         edx,edx
00000006: test        ecx,ecx
00000008: setg        dl
0000000B: push        esi
0000000C: mov         esi,dword ptr [esp+0Ch]
00000010: mov         eax,1
00000015: add         esi,edx
00000017: test        edx,edx
00000019: je          00000045
0000001B: push        ebx
0000001C: push        edi
0000001D: lea         ecx,[ecx]
00000020: imul        eax,ecx
00000023: mov         edi,ecx
00000025: xor         edx,edx
00000027: dec         ecx
00000028: test        ecx,ecx
0000002A: setg        dl
0000002D: mov         ebx,edx
0000002F: sub         ebx,esi
00000031: sub         ebx,eax
00000033: add         ebx,edi
00000035: test        edx,edx
00000037: mov         esi,ebx
00000039: jne         00000020
0000003B: mov         eax,dword ptr [esp+14h]
0000003F: pop         edi
00000040: pop         ebx
00000041: mov         dword ptr [eax],esi
00000043: pop         esi
00000044: ret
00000045: mov         ecx,dword ptr [esp+0Ch]
00000049: mov         dword ptr [ecx],esi
0000004B: pop         esi
0000004C: ret
```