Introduction

A piece of software is a tool that performs a particular task. The task may be narrow – calculate the sum of two numbers – or broad – compile source code to run on a supercomputer. In either case, the software operates according to its implementation. There is an understanding between a software developer and a software user: A program expects a certain type of input and, if the user gives the program input of that type, the program will compute the proper output. Contrariwise, the software developer assumes that the user will only expect the program to operate the way it was intended.

In the traditional world, when a developer releases a piece of software to a user, that developer is releasing everything necessary for the user to run that software on his/her computer. In order to operate the software, the computer needs to be able to understand its instructions. And if the computer can understand the instructions, then so can the user. In other words, when the developer releases his/her software, it is reasonable to assume that a (motivated) user could “reverse engineer” the software and decipher its mechanisms.

There are many reasons why software *developers* may want to prevent reverse engineering. The software may contain a proprietary algorithm or it may contain a secret code that will limit its use to authorized computers. And, though there are many industrial examples of techniques to accomplish this, none have proven immune to reverse engineering. Each technique is essentially an ad-hoc method for obscuring a program's inner-workings.

With the advent of cloud computing, developers have a new tool for masking the implementation details of their programs. Instead of releasing the software for a user to run on his/her computer, the software is accessed over the Internet. Inputs are sent and outputs are received remotely instead of locally.

On the other hand, there are also many reasons why *users* may want a program's implementation to be safe from prying eyes. Hiding or obscuring the implementation might might protect the user's identity or make it more difficult for a hacker to attack the software with malicious input. If an interloper cannot understand the detail's of a program's operation, they cannot *meaningfully* modify it to operate maliciously. In other words, with a hidden or obscured program implementation a user would feel safe knowing that the software has not been tampered.

For these reasons, among others, there is interest from both users and developers for a way to cloak or obscure software. The cryptographers studying this topic have settled on *obfuscation* as the term to describe their work.

The parallels between program obfuscation and message encryption are striking. In traditional message encryption, the goal is for a sender and a receiver to exchange messages with the guarantee that they are they are the only two people privy to the contents. In program obfuscation, the goal is for a developer to release a piece of software to a user without revealing anything about its implementation. The relationship between the two tasks is so close that one researcher co-opted the phrase “encrypting a message” and now refers to program obfuscation as “encrypting a functionality.” (Sahai)

Formalization

For a long time, cryptography (message encryption) was an *art*, not a science. Only when researchers and scientists began formalizing the definitions and stating assumptions did it became possible for

people to prove that a cryptographic scheme was secure. Until 2001, program obfuscation was an art, not a science. It took Barak et al. publishing a paper on the topic to get scientists to be precise about the meanings, definitions and assumptions behind program obfuscation. Unfortunately, this paper both began and ended inquiry into the field of program obfuscation. The same paper that introduced the first precise definition of obfuscation and defined an object that performed obfuscation (a so-called obfuscator) also proved that obfuscation was impossible.

Barak et al. defined an obfuscated program as a program that is the derivative of an original program with equivalent functionality. An obfuscator, denoted O, takes a program P as input and generates a obfuscated program P'. By definition, P' operates with at most polynomial-time overhead in speed compared to P and must compute the same function. These are the syntactic requirements of an obfuscated program. Depending on the class of obfuscation, there are additional criteria added to that definition.

In their paper, Barak et al. defined three different versions (strengths) of program obfuscation: virtual black box obfuscation, indistinguishable obfuscation and differing-inputs obfuscation.

Virtual Black Box Obfuscation

The intuition behind virtual black box obfuscation is that an obfuscated program should behave like a “black box” whose internals are off limits. The user of an obfuscated program must not be able to learn any more about P than its input/output characteristics even if they are allowed an infinite capacity to inspect O(P). In other words, what a user can learn about the original program P in the process of submitting arbitrary inputs and inspecting the corresponding outputs (oracle access) is exactly the same as what an adversary could learn about an obfuscated version of the same program O(P) given input/output access *and the ability to inspect its implementation*.

Although it initially seems counterintuitive to define obfuscation this way, it is similar to the Simulator Paradigm for Zero Knowledge proofs. In a Zero Knowledge proof system, a foe learns nothing new from a prover if there exists a simulator that can compute the same results as the prover. In virtual black box obfuscation, a foe learns nothing new about the obfuscated program if the original program is aped exactly by a “simulator” that has only access to the original program's inputs and outputs.

Unfortunately, as quickly as virtual black box obfuscation was formalized, it was shown to be impossible. (Barak) The authors proved that there exists a family of functions that are unobfuscatable. An *unobfuscatable* function f_u has a property (even 1-bit) that is easy to calculate given access to any implementation of f_u but is impossible to calculate accurately with only oracle access to f_u. In other words, this property can be “hidden” when access to f_u is restricted to f_u's oracle. However, anytime an attacker gets access to an implementation of f_u, this property is easy to compute. Recall that because obfuscations are functionally equivalent to the original, $O(f_u)$ is still an implementation of f_u and therefore an attacker given $O(f_u)$ will always be able to reveal f_u's secret property.

To construct such unobfuscatable function, it is useful to think the same way that the authors' think of a program. The authors conceive of a program as a “succinct description of [a] function” (Barak) or as a way to “compress” a function's truth-table. (Sahai) A *learnable function* is a function whose entire truth-table can be built by observing the pairs of inputs to a function and the outputs it generates. For a learnable function, there is no knowledge gap between a user who gets access to a black box that computes f and a user who gets access to a black box that computes f *and* the ability to inspect its implementation. The authors base their construction of the unobfuscatable function around the idea that

“… there is a fundamental difference between getting black-box access to a function and getting a program that computes it.” (Barak)

As with other great advances in theoretical computer science, the authors' construction is a self-referential and pathological. The function is impossible to compute with only oracle access. The proof is complex but starts with the definition of a simple “function that cannot be exactly learned with oracle queries.” (Barak) Let C be a function

$$C_{\alpha,\beta}(x)=\begin{cases}\beta, \textit{if } x=\alpha \\ 0^k, \textit{otherwise}\end{cases} \textit{where} \alpha,\beta,x\in\{0,1\}^k$$

The authors' final construction of an unobfuscatable function is more complex than this simple example, but the idea is the same. Sahai put it colorfully when he described these functions as “self-eating.” (Sahai)

The authors are not prepared to qualify how important this unobfuscatable construction is for actual practitioners. They allow for the possibility that even with this impossibility result, the virtual black box definition of program obfuscation may be useful. However, the result is critical because it shows that virtual black box program obfuscation is impossible in the strictest sense.

Indistinguishable Obfuscation

Lost in the excitement of their original work was the authors' definition of a second type of obfuscation: indistinguishable obfuscation (iO). Indistinguishable obfuscation adds to the syntactic requirements of obfuscation this third criteria: an adversary cannot tell the difference between an obfuscation of two programs that compute the same functionality. More precisely, no polynomial-time adversary can distinguish between $O(C_1)$ and $O(C_2)$ where C_1 and C_2 are chosen at random from a class of circuits C that compute the same function. In an *iO guessing game*, the adversary would choose two circuits C_1 and C_2 from C and provide them to a challenger. The challenger would choose one of the two circuits to obfuscate and return that obfuscated program $O(C_?)$. The adversary's output would be a single bit that is its guess as to which circuit was obfuscated. For all probabilistic polynomial-time adversaries A_{iOGG},

$$\left|P[A_{iOGG}(C_1)=1]-P[A_{iOGG}(C_2)=1]\right|\leqslant\epsilon.$$

Even as researchers searched for an obfuscator that would meet such a definition, they also wondered if it would be useful. In other words, would software developers/users or cryptographers gain any actual power from such an obfuscator? In 2012, Garg et al. simultaneously demonstrated the power of such an obfuscator and one potential candidate algorithm.

Garg et al. showed that iO reduces to functional encryption. In a functional encryption scheme (Gen, Dec, Enc, F), a user gives Gen the parameter *x*. Gen returns a string *y* and a secret key that can decrypt *y*. However, when the user decrypts *y* using the secret key, they are able to learn only the value F(*x*) and nothing about *x* itself. Even with collusion among keyholders, there is nothing that can be learned about the input values *x*.

The construction of their candidate obfuscator is quite magical. It relies on three key components and the introduction of new computational hardness assumptions. It first relies on a theorem of Barrington from the 1980s that shows that any program in a particular computational class (NC^1) can be converted to the result of a multiplication of values picked from the permutations of specially constructed matrices. (Garg) The authors' combine Barrington's theorem with Kilian's protocol which allows two

players (Alice and Bob) to “evaluate an NC^1 circuit on their joint input.” (Garg) In the context of this iO construction,

- the circuit is the universal circuit,
- Alice is the (imaginary) obfuscator and her half of the input is the program to obfuscate, and
- Bob is the obfuscated program user and his input is the obfuscated program's input.

This novel application of Barrington and Kilian's result is the authors' first attempt at an indistinguishability obfuscation scheme.

However, there are problems with using “Kilian's protocol as-is.” (Garg) The most pernicious problem is that Bob may learn something about the obfuscated circuit if his input does not obey the semantics of the protocol. To prevent such an attack, the authors apply multilinear jigsaw puzzles to their scheme. *Multilinear jigsaw puzzles* are like multilinear maps. *Multilinear maps* are an abstraction of bilinear maps that are themselves an abstraction of groups. A *bilinear map* is a mapping function $e:G \times G \rightarrow G_T$ such that $e(g^a, g^b) = e(g, g)^{ab}$ for $g \in G$. (Boneh) If $e:G_1 \times G_2 \rightarrow G_T$ and $G_1 \neq G_2$, e is called an *asymmetric bilinear map*. (Boneh) A multilinear map is a function $e:G_1 \times G_2 \times G_3 \ldots G_n \rightarrow G_T$ such that $e(g^{a1}, g^{a2}, \ldots, g^{an}) = e(g_1,g_2, \ldots, g_n)^{(a1 \bullet a2 \bullet \ldots \bullet an)}$ for $g_i \in G_i$. According to the authors, multilinear maps are subject to certain cryptanalytic attacks that make them unsuitable for this application. Instead, they use multilinear jigsaw puzzles that are multilinear maps whose elements can only be recombined to find element g^0 for $g \in G_T$. When the elements are combined in the proper way, their combination can be verified but nothing else about the elements can be learned. Once these jigsaw puzzles are applied to the Barrington matrices, the original program is indistinguishably obfuscated.

While the mathematical properties of multilinear jigsaw puzzles elements are useful in this construction, they represent a new computational hardness assumption. If combining jigsaw puzzle entries is not hard, then their obfuscator does not satisfy the definition of iO. Garg et al. have subjected their assumption to cryptanalytic techniques and do not see a reason to believe that the problem is easy. (Sahai)

Once the practical limitations on such a construction have been resolved, it will be amazing to see how this affects the implementation of software security systems. In the meantime, see Randomization and Parallel Execution for a practical system that may approximate iO.

Differing-inputs Obfuscation

Barak et al. proposed yet a third definition of obfuscation: differing-inputs obfuscation. This type of obfuscation adds to the syntactic requirements the criteria that, given any two obfuscated programs, if there is an an adversary that can distinguish the two, there must exist another adversary that can find an input on which the output of the two circuits differ. Again, this criteria is added to the baseline set of the syntactic requirements of obfuscation.

Differing-inputs obfuscation is a stronger definition of program obfuscation than iO. It is easy to see how differing-inputs obfuscation implies iO. In iO, the adversary considers circuits C_1 and C_2 randomly selected from a class of circuits that each compute the same function. Therefore, by the syntactic requirements of program obfuscation, programs $O(C_1)$ and $O(C_2)$ compute the same function and therefore no single input to $O(C_1)$ and $O(C_2)$ will produce different outputs.

Though Barak et al proposed this as a definition and researchers have constructed useful tools using it as a primitive (Ananth), most believe that such an obfuscator does not exist. (Garg 2014) In *On the Implausibility of Differing-Inputs Obfuscation and Extractable Witness Encryption with Auxiliary*

Input, Garg et al. write that, although no definite impossibility proof exists, a “surprising consequence” of the existence of a different-inputs obfuscator makes it highly unlikely that one exists.

Obfuscation in Systems Design

As described above, Garg et al. have constructed an obfuscator that meets Barak et al.'s definition for iO. Although their construction is efficient in a computationally theoretic way, it is not yet efficient in a practical sense. Therefore, systems designers and implementers remain focused on other ways to use obscurity and randomness to secure systems.

Systems designers look at software security very narrowly, according to a threat model. A *threat model* defines a specific method that an enemy may use to attack a system. It also gives assumptions about the power of the adversary and the power of the defender. Then, system designers create defenses that protect against attackers operating under these assumptions and say that their system is secure. So, in contrast to the general contract implicit between developer and user described in Introduction, systems designers call software secure with respect to a certain threat model.

Researchers and security professionals often focus on the remote attacker threat model. In this model, an external entity, the attacker, wants to compromise a system by breaking a piece of software. The attacker does not have access to *the* running copy of the program but is assumed to have access to a *version* of the program that he/she can inspect at any level of detail. Moreover, “the attacker understands the protection methodology and may have access to [similar or the same] tools for applying … protections.” (Hiser) The attacker interacts with the software through its external interfaces. For example, under this model, an attacker targeting Adobe Reader

1. has access to an implementation of the PDF reader,
2. can craft a (malicious) input
3. knows the mechanisms that the user has employed for protection
4. but does not have access to the actual version of the software running on the user's computer.

The crafty attacker may use any means of attacking the target software. In most cases, successful attacks take advantage of the fact that exploitable instructions exist at certain places in the target. Many operating systems and hardware manufactures now ship with defenses against this type of attack but they are not completely effective. (Hiser) Below are descriptions of two practical state of the art mechanisms that use randomness and obfuscation to secure systems against the remote attacker threat model.

Cryptography and Software Dynamic Translation

Software dynamic translation (SDT) is a technique for translating a program at runtime with nothing more than access to its machine code. SDT employs a runtime engine that interrupts the traditional fetch-decode-execute cycle by adding an additional decoding step. In this extra decoding step, the SDT runtime may modify an instruction in an arbitrary manner. SDT is often used to retarget a program from one platform to another (x86-32 to ARM), optimize a program for a particular execution environment (CPU type, cache size, disk performance) or add error handling (off-by-one, buffer overflow, etc).

Leveraging the ability of SDT runtime engines to modify an instruction before it is executed, designers have come up with instruction set randomization (ISR) schemes. In ISR, “[e]ach byte of protected code in the program is individually scrambled using pseudorandom numbers seeded with a random key that is unique to each program execution.” (Barrantes) During execution, the SDT runtime engine de-

randomizes the instructions and submits them to the processor for execution. If the SDT runtime engine fails to properly de-randomize an instruction, the implication is that the instruction was not part of the original program – it was somehow inserted from the outside. In this case, the runtime can protect against a remote code injection attack by disregarding that instruction.[1]

Because there are only a limited number of valid instruction types in a particular instruction set and only so many valid parameters for those instructions, the amount of entropy in a system that randomizes per instruction is low. Given enough time, an attacker may be able to determine the random key and ultimately create a malicious input that contains remote code that the SDT runtime engine properly de-randomizes and executes.

As a countermeasure, Hu et al. developed an encrypted ISR system that encrypts blocks of instructions. In this system, each 1K of instructions is encrypted with AES. The SDT runtime engine is given the corresponding decryption keys to use during execution. A runtime, the code is decrypted block-by-block before being passed on to the processor. In this system, the program's binary is randomized *and* encrypted. Given the assumptions of the remote attacker threat model, the attacker is assumed to have access to the program's implementation. Besides seeing a program whose instructions are somehow permuted (as in the pseudorandom ISR implementation), an attacker of this system would see a program whose instructions are encrypted – a significant improvement.

Such a system is useful in a wider context, too. Because the program itself is encrypted, it can only be executed under an SDT runtime engine with the appropriate decryption keys. The utility of this is broad. Unfortunately, it is still not as useful as a system that obfuscates program's according to Barak et al.'s definitions. Even in encrypted ISR, the user's system *knows* more about the program than its inputs and outputs. When the SDT runtime engine decrypts the instructions, an eavesdropper can record the sequences that are sent to the processor for execution.

Randomization and Parallel Execution

Defense in depth is a common security technique used by system designers. They believe that layering defenses one on top of another raises the level of security of a system because a successful attack will have to compromise each security provision simultaneously. Cox et al. have described a defense in depth system called N-variant systems for protecting hosts that run Internet-attached software. Internet-attached software encompasses a wide set of programs but the authors are mostly concerned with servers – web servers, database servers, application servers, etc. In the context of the remote attacker threat model, this system attempts to protect against an attacker who wants to gain access to or maliciously modify the underlying host.

In an N-variant system, there are no secrets. The attacker knows everything about how a system operates and its protection. The attacker knows how a system operates and the system's inputs and outputs are not a secret. To protect itself, the host generates a set V of N different versions of the same program and runs them simultaneously, in lock-step. Every external input is replayed to all $v \in V$. Only when every variant operates identically on an input is the system allowed to generate an output. This output might be a response to the user or it might be a modification of the underlying host (disk access, database operation, etc). Therefore, to successfully attack such a system, an adversary must find a single malicious input that simultaneously compromises each of the N variants. The authors of the

1 In remote code injection, an attacker introduces new code for the program to execute through their malicious input.

paper believe that this improves the security of a system.

It seems that there are parallels between an N-variant system and iO. Each of the N-variants, like each C_i selected from a functionally equivalent class of circuits C, compute the same function. Because the attacker can only observe the system's inputs and outputs (it is located across the Internet, after all), they essentially have oracle access to the function. In an N-variant system, it is as if the attacker is interacting with *one* randomly chosen functionally equivalent implementation of the program at the same time that they are interacting with *every* functionally equivalent implementation. Therefore, by definition, for any adversary A_{iOGG} that competes in the iO guessing game,

$$\left|P[A_{iOGG}(v_1)=1]-P[A_{ioGG}(v_2)=1]\right| \leqslant \epsilon \; for\; all\; v_{1,}\, v_2 \in V$$

where *V* is the set of variants.

Future Work

In the future, it would be interesting to study intersection of cryptographic program obfuscation and return oriented programming (ROP). In *ROP*, a program is turned against itself and used as a source of code to perform operations that the developer did not originally intend. In other words, in the hands of a skilled attacker using ROP, a program is not necessarily functionally equivalent to itself. In fact, researchers have shown that most programs contain enough “gadgets” to execute Turing-complete algorithms. (Shacham) The fact that, given a precise set of inputs, a program may be tricked into calculating something entirely different (and with no computational limit) may invalidate some assumptions of iO.

Conclusion

Like cryptographers studying message encryption in the late 20th century, theoreticians studying program obfuscation have made the transition to modernity. They are working with precise definitions and explicit assumptions about program obfuscation. Although systems designers still treat program obfuscation as an art, they do speak precisely and formally about the definition of system security. There are plenty of additional opportunities for program obfuscation researchers and system designers to collaborate and make the theoretically possible practically doable.

References

Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, Mark Zhandry. "Differing-Inputs Obfuscation and Applications.". *IACR Cryptology ePrint Archive*. 2013, vol 2013, p. 689.

Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, Ke Yang. *On the (Im)possibility of Obfuscating Programs*. Kilian, Joe. Springer Berlin Heidelberg, 2001, 1-18. (Lecture Notes in Computer Science). (2139). http://dx.doi.org/10.1007/3-540-44647-8_1. ISBN: 978-3-540-42456-7.

Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security* (CCS '03). ACM, New York, NY, USA, 281-289. DOI=10.1145/948109.948147 http://doi.acm.org/10.1145/948109.948147

Dan Boneh. “Pairing basics.” 3rd Bar-Ilan Winter School on Cryptography. Bar-Ilan University, Tel-Aviv, Israel. Feb. 5, 2013. Lecture.

Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (USENIX-SS'06), Vol. 15. USENIX Association, Berkeley, CA, USA.

Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, Brent Waters, "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits," *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on* , vol., no., pp.40,49, 26-29 Oct. 2013 doi: 10.1109/FOCS.2013.13

Sanjam Garg, Craig Gentry, Shai Halevi, Daniel Wichs. *On the Implausibility of Differing-Inputs Obfuscation and Extractable Witness Encryption with Auxiliary Input*. Garay, JuanA. and Gennaro, Rosario. Springer Berlin Heidelberg, 2014, 518-535. (Lecture Notes in Computer Science). (8616). http://dx.doi.org/10.1007/978-3-662-44371-2_29. ISBN: 978-3-662-44370-5.

Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (SP '12). IEEE Computer Society, Washington, DC, USA, 571-585. DOI=10.1109/SP.2012.39 http://dx.doi.org/10.1109/SP.2012.39

Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments* (VEE '06). ACM, New York, NY, USA, 2-12. DOI=10.1145/1134760.1134764 http://doi.acm.org/10.1145/1134760.1134764

Amit Sahai. “How to Encrypt a Functionality.” Quantum Games and Protocols. Simons Institute for the Theory of Computing, Berkeley, CA. Feb. 25, 2014. Lecture.

Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (CCS '07). ACM, New York, NY, USA, 552-561. DOI=10.1145/1315245.1315313 http://doi.acm.org/10.1145/1315245.1315313