Integrated Software Fingerprinting via Neural-Network-Based Control Flow Obfuscation

Haoyu Ma*, Ruiqi Li*, Xiaoxu Yu*, Chunfu Jia*, and Debin Gao†
*College of Computer and Control Engineering, Nankai University, Tianjin, 300071 China
†School of Information Systems, Singapore Management University, Singapore, 188065 Singapore

***Abstract*—Dynamic software fingerprinting has been an important tool in fighting against software theft and pirating by embedding unique fingerprints into software copies. However, existing work uses methods from dynamic software watermarking as direct solutions in which secret marks are inside rather independent code modules attached to the software. This results in an intrinsic weakness against targeted collusive attacks since differences among software copies correspond directly to the fingerprint-related components. In this paper, we suggest a novel mode of dynamic fingerprinting called integrated fingerprinting, of which the goal is to ensure all fingerprinted software copies possess identical behaviors at semantic level. We then provide the first implementation of integrated fingerprinting called *Neuroprint* on top of a control flow obfuscator that replaces program's conditional structures with neural networks trained to simulate their branching behaviors [1]. Leveraging the rich entropy in the outputs of these neural networks, Neuroprint embeds software fingerprints such that a one-time construction of the networks serves both purposes of obfuscation and fingerprinting. Evaluations show that due to the incomprehensibility of neural networks, it is infeasible to de-obfuscate the software transformed by Neuroprint or attack the fingerprint using even the latest program analysis techniques. Revealing information regarding the hidden fingerprints via collusive attacks on Neuroprint is difficult as well. Finally, Neuroprint also demonstrates negligible runtime overhead.**

***Index Terms*—Software fingerprinting, code obfuscation, neural network.**

I. Introduction

With software industry growing rapidly, software developers become to realize that more attention should be paid on software security issues regarding potential intellectual property violations. On one hand, the increasingly powerful reverse engineering techniques have made software verification and malware analysis more effective [2], [3]. Yet on the other hand, they also unfortunately enable the so called *man-at-the-end (MATE) attacks* [4] in which the adversary is assumed to possess full control of the system and thus could thoroughly inspect and analyze the running programs for purposes like software theft, pirating, and cracking. With state-of-the-art automatic software analyzing and testing techniques [5]–[8], MATE adversaries are now not only able to dissemble and analyze the code text of a tested software statically, but can also dynamically monitor its execution. This allows an attacker to reveal or even modify the software's internal logic for purposes like evading authorization checks, plagiarizing valuable modules, etc. Discouraging software theft and pirating includes several security goals, out of which an important one is to establish *virtual uniqueness* on software's copies to be distributed. Such uniqueness makes it possible to identify the authorized user of any specific copy of a software so that perpetrators can be found after the fact of proprietary right violation. To achieve such an objective, *software fingerprinting* technique has been considered as a promising strategy.

Corresponding author: Chunfu Jia (email: cfjia@nankai.edu.cn).

A. Dynamic Software Fingerprinting and Collusive Attacks

Software fingerprinting is generally known as a derivative form of *software watermarking*. It typically requires to embed a secret signature into the subject software such that a concerned authorized party can reliably recognize it. Formally, a software fingerprinting scheme $\mathbf{F} : \{E(\cdot), R(\cdot)\}$ consists of two major portions: the embedding process $E(\cdot)$, which is by all means a program transformation, and the recognition protocol $R(\cdot)$. Given a subject program $\mathcal{P}$, a fingerprint f, and the secret input (or parameter) i as f's trigger, $\mathbf{F}$ produces the corresponding fingerprinted program instance $\mathcal{P}_f$ by

$$E(\mathcal{P}) \xrightarrow{f,i} \mathcal{P}_f. \tag{1}$$

Later, using the same i, a recognition process should be able to reliably output f from $\mathcal{P}_f$ by computing

$$R(\mathcal{P}_f) \xrightarrow{i} f. \tag{2}$$

Due to the relation between software watermarking and fingerprinting, previous works tend to believe that the requirement of fingerprinting can be sufficiently addressed using direct transplantation of the existing watermarking schemes. Hence software fingerprinting systems, as their watermarking relatives, are also in either the *static* [9] or *dynamic* manner [10], [11]. The difference between these two flavors is mainly on whether the fingerprints are embedded as special features of static program texts, or runtime-generated execution states/data objects. This paper will focus on dynamic software fingerprinting only, given that it is widely accepted as the better solution among the two options regarding to the effectiveness against attacks based on semantic-preserving transformations [10]–[14].

Again, we emphasize that software fingerprinting techniques are usually used for the purpose of tracing perpetrators after the protected software has been pirated. For example, when Alice discovers a pirated copy of a software of hers, she then applies her fingerprint extractor to discover who purchased the original copy. Due to this usage scenario, the fingerprint

embedded has to be unique for each individual copy of the software. This makes software fingerprinting different from software watermarking. What is often neglected (yet potentially fatal) is: because of the difference on their scenarios, solutions proposed for watermarking become rather vulnerable when applied to fingerprinting.

Existing dynamic watermarking approaches work by formatting the secret signature to be embedded as outcomes of certain payload code and then attach the payload code at one or multiple position(s) in the subject software [9]–[11]. Let $\mathcal{F}$ represent such a payload code corresponding to a fingerprint f, then the embedding process $E(\cdot)$ in these designs works by simply putting $\mathcal{F}$ together with the subject program after creating it, i.e.

$$\mathcal{P}_f = \mathcal{P} + \mathcal{F}. \tag{3}$$

In other words, $\mathcal{F}$ here is built to work *only* for watermarking purpose, its execution has basically no affect on the software $\mathcal{P}$'s original functionality. Obviously, after the watermarking schemes of this type are transplanted to server in the software fingerprinting scenario, such characteristic remains true.

Problem is, being a piece of program itself, an independent fingerprint module defines its own semantics. Let $\Theta(\mathcal{X})$ be the structural operational semantics of program $\mathcal{X}$, it is defined by a triple $\langle \Sigma(\mathcal{X}), \imath(\mathcal{X}), \tau(\mathcal{X}) \rangle$ in which

- $\Sigma(\mathcal{X})$ is the set of all possible internal (both memory and control) states of $\mathcal{X}$;
- $\imath(\mathcal{X}) \subseteq \Sigma(\mathcal{X})$ is the set of all possible initial states that $\mathcal{X}$ may start with; and
- $\tau(\mathcal{X}) \subseteq \Sigma(\mathcal{X}) \times \Sigma(\mathcal{X})$ is the set of all possible transition relations in $\mathcal{X}$.

To express f via executing the payload $\mathcal{F}$, the method taken by nowadays' approaches is to define a mapping from f to:

- a trail of transition relations $\tau_f = \{\tau_0, \tau_1, \cdots, \tau_k\}$ (in which $\tau_{x\in[0,k]} \in \tau(\mathcal{F})$) to be found in $\mathcal{F}$'s behavior; or
- the combination of internal states $\sigma_f = \{\sigma_0, \sigma_1, \cdots, \sigma_k\}$ (in which $\sigma_{y\in[0,k]} \in \Sigma(\mathcal{F})$, and $\tau_{x\in[0,k]} = (\sigma_{x-1}, \sigma_x)$) that is in corresponding to τ_f.

In either case, f is seen as a straight-forward interpretation of $\mathcal{F}$'s semantics (or at least part of them), i.e., $f \subseteq \Theta(\mathcal{F})$. As the result, given any two fingerprints $f_1 \neq f_2$ as well as the corresponding program instances $\mathcal{P}_{f_1}, \mathcal{P}_{f_2}$ where they are embedded,

$$\Theta(\mathcal{P}_{f_1}) = \Theta(\mathcal{P}) + \Theta(\mathcal{F}_1) \neq \Theta(\mathcal{P}) + \Theta(\mathcal{F}_2) = \Theta(\mathcal{P}_{f_2}). \tag{4}$$

Therefore, for a so fingerprinted program, the individualized instances of it would demonstrate semantic-level differences, which indicate nothing else but the fingerprint payload code. This makes the adversary's task of locating fingerprint-related code much less challenging, since once he possesses multiple fingerprinted copies of it, he only needs to look into the semantically different portions to find the fingerprint payloads (and this simple way of disclosing the supposed-to-be-hidden fingerprint is known as the *collusive attacks*). We believe that this inherent feature degrades the effectiveness of existing dynamic fingerprinting schemes and makes collusive attacks a critical threat to the technique.

B. Obfuscation Enhanced Software Fingerprinting

Being vulnerable to collusive attacks raises yet a secondary problem for dynamic software fingerprinting: the compromised resilience to attacks aiming to remove the fingerprint, or by all means make it unrecognizable (which are known respectively as the *subtractive attacks* and the *distortive attacks*). The basic security assumption of dynamic watermarking/fingerprinting approaches against these attacks lies in that any transformation that preserves semantics of *both* the subject software and the fingerprint payload will not compromise the fingerprint. However, due to the lack of inter-dependency between these two components, it becomes unnecessary for the attackers to care about the integrity of fingerprint payload given that the subject software would still be fully functional without it. This makes the payload code of existing dynamic fingerprinting designs rather fragile upon spotted, which is even worse with the presence of collusive attacks.

One possible solution to enhance the security of current dynamic fingerprinting approaches is to additionally apply *code obfuscation* on fingerprinted software. According to Palsberg et al. [15], the goal is to

- make the fingerprint payload code unrecognizable; or
- disguise it in a way so similar to the subject software, that tampering with it seems to be endangering the software's integrity and/or correctness.

That said, to our best knowledge, the only existing obfuscation enhanced dynamic watermarking/fingerprinting design is the *Droidmarking* system proposed in [14]. This approach turns its payload snippets into the so called *self-decrypting code* (first proposed in [16]), such that they are made statically unreadable. Only when the protected software takes certain branches (where such code snippets are placed) at runtime will the obfuscated code blocks be correctly decrypted and become functional. However, since the so constructed payload snippets are still independent to the subject software, Droidmarking has to accept a significant trade-off between stealth and resilience. In fact, it is claimed to be completely "non-stealthy". Even so, each of the Droidmarking's code snippets is still guaranteed to be unprotected status upon being reached and revealed into functional executable at runtime, afterward they can be safely removed.

C. Contributions of This Paper

As explained above, for the existing works on dynamic fingerprinting, the fundamental idea of payload code's work mode has become a source of problems. Therefore, when it comes to exploring a better solution, one question naturally raises: *is it possible to merge dynamic fingerprints into software's original functionality*? Motivated by the question, this paper proposes a novel dynamic fingerprinting mode called *integrated fingerprinting*. Our new model suggests that the embedding of software fingerprint should be a semantic-preserving transformation on the subject software so that the collusive attack to perform differential analysis on the fingerprinted instances would not work. This transformation turns software's original execution into an equivalent yet special new form, which, under a pre-determined situation, exhibits

the fingerprint information as runtime side effects. The goal of integrated fingerprinting mode is to bond the fingerprint to the carrier software's integrity. Thus under this mode, to isolate or erase the embedded fingerprint while keeping the fingerprinted software itself functional becomes difficult.

To take one step further, we propose the first realization of integrated fingerprinting – named *Neuroprint* – on top of a previously proposed neural-network-based software control flow obfuscation approach of ours [1]. Our underlying obfuscator protects software's control flow integrity by simulating branching logic of its conditional structures with neural network computation. Specifically, our trained neural network outputs offset of destination code blocks of the protected conditional branch according to the value of its input variable(s), then use the offset to perform the replaced control transfer correctly. In addition to the obfuscation, over all the protected conditional branches, our Neuroprint system selects a relatively small set of them on a particular execution trail (which corresponds to the user-specified secret input, henceforth the *fingerprint trail*), and inject fingerprint in the selected branches. Neural networks for obfuscating these branches are trained to "remember" not only control information, but also fractions of the fingerprint message. Later, upon being invoked on this trail, these special networks release the fractions they carry via their computation results (the same ones that also provide control offsets for the obfuscated branches) in the same time of performing the obfuscated control transfers. Neuroprint's recognizer can therefore reconstruct the complete fingerprint by monitoring the execution procedure and collecting the released fractions.

Intuitively, the well-known complexity in comprehending the rules represented by neural networks ensures that whatever knowledge they are trained with (including both the branching logic and fingerprint information) cannot be explained using symbolic rule combinations of a reasonable scale [17]–[19]. Meanwhile, by making the neural networks "multi-tasking", our approach turns the fingerprint into an inner component of carrier software's (obfuscated) conditional branches, thus manages to avoid using independent payload code. The result of these together is that it is made impractical for Neuroprint's adversaries to

- on semantic-level, reliably distinguish the fingerprinted branches from those obfuscated-only ones; or
- with an acceptable cost, recover any of the obscured control transfers via automated reverse-engineering techniques based on reasoning systems.

In other words, to compromise a fingerprint embedded using Neuroprint is at least as difficult as to perform a comprehensive de-obfuscation on the transformed software, which is next to impossible as well. We evaluated our Neuroprint design on several aspects, including its strength, the validity of fingerprint recognition, and performance. Results indicate that Neuroprint effectively resists major attack scenarios targeting either control obfuscation or software fingerprinting while introducing acceptable overhead.

As an extension of [1], this paper makes a more comprehensive exploration on applying the intrinsic security value neural network into a new software security scenario other than code obfuscation, namely the designing of software fingerprinting. We for the first time propose the integrated fingerprinting mode to be a novel solution that may address the major drawbacks of existing dynamic fingerprinting approaches. To this end, in our Neuroprint system we re-define the output formatting of neural networks in the original obfuscator to include a fingerprint section along side the control data of obfuscated branches. Neuroprint also establishes a self-authenticated fingerprint recognition protocol to ensure the correct extraction of the fingerprint. The amendments create a way of dynamically carrying fingerprint messages within program's own semantics via transformations difficult to be undone, which has not been accomplished before.

II. Background on Neural Network

Artificial neural network (e.g. feedforward neural network, like demonstrated in Fig. 1) is a connectionist model consisting of an interconnected group of artificial neurons. It is known to be a distributive and fault-tolerant system capable of representing non-linear algorithms with powerful computational capability. During the 1990s, several researches suggested that a neural network of the above representative model is a universal function approximator and should be able to simulate arbitrary functions [20], [21]. Researches on *deep learning* have also shown that networks with more hidden layers could be even more powerful [22], [23]. In this paper, we show that neural network also provides solid security properties in software protection.

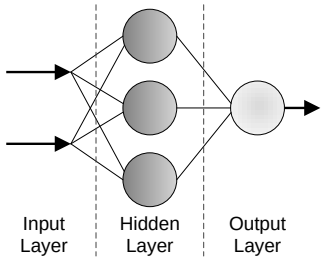


Fig. 1: An example of feedforward neural network [1].

A. Rule Extraction from Neural Network

Researchers generally believe that a main weakness of neural network is the absence of a capability to explain either its process of arriving at a specific decision/result, or in general, the knowledge embedded in it in human-comprehensible form [17]. In 1996, Golea studied the intrinsic complexity of rule extraction from neural network and came out with key results that the following two problems are both NP-hard [18], [19].

- Extracting the minimum Disjunctive Normal Form expression from a trained (feed-forward) neural network; and
- Extracting both the best monomial rule and the best M-of-N rule from a single perceptron within a trained neural network.

To the best of our knowledge, existing rule extracting solutions could at most generate relatively large combinations

of symbolic rules in approximating the knowledge embedded in neural networks [24], [25]. Some approaches work only on networks of limited types [26]. Also, we find that for scenarios like code obfuscation and software watermarking/fingerprinting, understanding of logic embedded in neural networks could be not just unnecessary but unwanted. In such case, the incomprehensibility of neural networks becomes an valuable feature to build security approaches on.

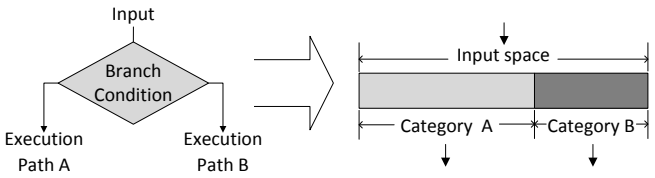


Fig. 2: Conditional execution and binomial classification

B. Training of Neural Networks and its Randomness

Since extracting rules represented by a neural network is typically hard, the general usage of neural network is to construct one that represent some given logic. This construction process is usually called *training*. Neural network training typically starts by assigning network parameters (i.e. the weight and bias factors) with some initiate states, then progressively adjusts their values with the goal of reducing the difference between output distribution of the network and that of the training sample to an acceptable degree. Since the network is of distributive structures, the training algorithms are "greedy", i.e. they find the best local solutions for each neuron rather than trying to achieve global optimum. Consequently, they require that the network be initialized randomly in order to avoid being stuck in a local minima.

The random initialization inevitably brings a side effect to neural network training, which is the obscure and indeterminate relationship between a neural network's explicit properties and its behavior. This is in fact rather obvious since even with the same training sample and network topology, every single training of the neural network with a different initialization vector would result in a solution with a distinct parameter set. We make use of this randomness property in our proposal of software fingerprinting, i.e. the Neuroprint, to fight against collusive attacks, which is further explained in Section IV-A.

III. Neural-Network-based Control flow Obfuscation [1]

Before bringing out the new dynamic fingerprinting design, we first present the basic unit of our system, namely the neural-network-based control flow obfuscation.

In programming, conditional logic is used to selectively transfer control to multiple execution paths based on the result of evaluation on some input. We, on the other hand, provide an equivalent interpretation of such logic as shown in Fig. 2. In this interpretation, a conditional operation is considered a *binomial classification* task where all possible values of the input space are classified into two groups, each corresponding to a determined execution path. Any particular input is examined and classified into one of the groups, then the program's control is directed to the corresponding path.

As there are various mathematical models that can precisely represent a binomial classification, this suggests that it is possible to choose one that provides some desirable security property of control flow obfuscation. We choose neural network as the candidate to build our obfuscator on, not only because it is a well-understood model in classification, but also due to the incomprehensible nature of its reasoning procedure. The extreme complexity of extracting rules embedded inside neural networks could help providing resistance against reverse engineering techniques that aim to reverse engineer the embedded logic of the obfuscated program routines.

A. Indirect Conditional Transfers

Constructing a neural network to represent the logic of the conditional transfer is relatively straight forward by directly applying existing training algorithms of neural network. However, doing so does not hide the behavior of the control transfer. That is, an attacker would still be able to tell the fact that there is a control transfer, and even the target of the transfers, although he fails to uncover the trigger condition. This is true for most existing obfuscating work in control flow obfuscation [16], [27].

Considering the capability of neural network, we believe we can do better — to even obfuscate the control transfer behavior. A conditional branch usually decides whether to direct control flow to a certain code block or to stay on the current code stream. The address of the instruction to be executed when a branch is taken is usually represented by a relative offset to the value of the instruction pointer. Given that neural networks are powerful enough to "remember" any predefined output values assigned to each group in the classification, it is possible to train the network to output such offsets directly (instead of outputting a boolean value in most existing obfuscating work) and to effectively use the neural network as a *conditional dispatcher*.

As shown in Fig. 3, our trained neural network outputs either zero or the corresponding offset of the code block, which is used to calculate the return address for the conditional dispatcher when it finishes its execution. Our design of the conditional dispatcher therefore maintains semantic equivalence with the original program. However, the control flow from the dispatcher to one of the code blocks (branch B as shown in Fig. 3) is now obfuscated and invisible to most automatic analysis tools.

Fig. 4 shows a simple program before and after we obfuscated the conditional transfer `if (y == 5)`. Recall that our objective is to obfuscate not only the branch condition (`y == 5` as in Fig. 4a) but also the control transfer (`jnz short loc_40100C` as in Fig. 4b).

Function `Conditional_Dispatcher()` (of which the detail implementation is given in Section V-C) replaces the original branch logic which uses a neural network built from function `Network_Update()` to maintain the same control behavior. As the result of our obfuscation, instruction

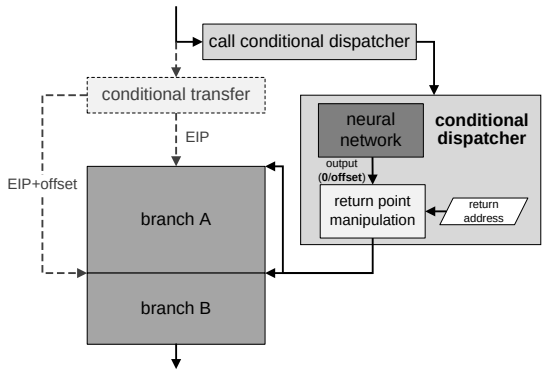
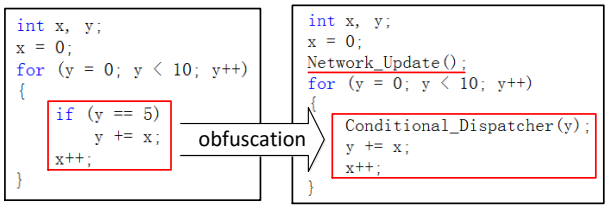
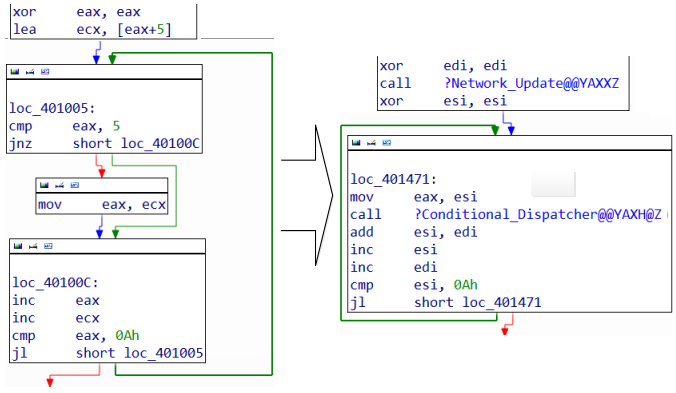


Fig. 3: Conditional dispatcher using neural network

`y += x;`, which is to be executed under certain condition, now looks as if it is in the same code block as `x++;`. The binary code of the corresponding loop and its obfuscation in Figure 4b shows the same structural misunderstanding to an analyzer caused by our obfuscating.



(a) Source code



(b) Binary code

Fig. 4: Example of control flow obfuscating

B. Applying Integer Neural Networks

One of the most important steps in the obfuscating is to construct the neural network. Traditional neural network uses sigmoid functions as neuron activators. However, due to the uniqueness of sigmoid functions and the high precision weight values assigned for network connections, these neural networks are relatively easy to detect and recognize. This potentially makes the embedded network easy targets to be located or traced. Therefore, we choose to use integer neural network to make the corresponding instructions for neural network calculation less suspicious.

Integer neural networks limit their weights to integers only, and apply simple step functions (which outputs 1 for a non-negative input or -1 otherwise) as their neuron activator [28], [29]. Due to the simple instruction profile in constructing an integer neural network, it is also easier to diversify its implementation.

Note that the input layer of integer neural networks is able to process numerics of any valid type given to it (but the weight factor of these nodes are still integers). So long as all the input values are numeric, activator in the input nodes would be able to properly map the input space to a enumeration of integer values (e.g. $\{1, -1\}$ in case of step functions), such that the neural network would then work entirely on integers starting from the subsequent hidden nodes. For branches with non-numeric conditions, a transformation process can be inserted before them to map the vector of its conditional variable(s) to a numeric space, then feed neural networks with the image vector. Thanks for the parallel structure of neural networks, branches with compound conditions can be directly processed without extra simplifying. Therefore, using integer neural networks in our system does not mean that candidate branches possible to be obfuscated are limited in any way.

C. Dynamic Network Construction

Obfuscating a program with neural network requires storing the weight matrix of the network for computing. To make it more difficult for an attacker to locate the matrix for analysis, we store it as heap-allocated objects with pointer aliasing [30], [31]. As shown in Fig. 5, data used by the neural network is created and updated dynamically.

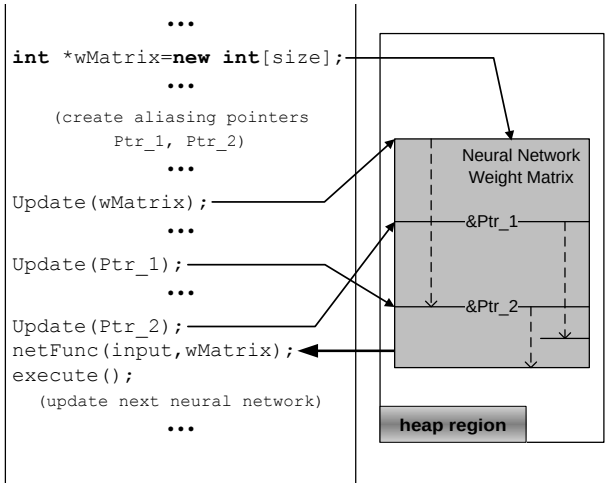


Fig. 5: Dynamic constructing and updating data for the neural network

We first allocate a memory region for the weight matrix, and then create pointers targeting different positions in the region to establish complex aliasing effect. After that, a set of instructions is deployed into the software's code sections to progressively update the allocated memory region (targeted by the pointers) with their operands, until finally the neural network matrix is constructed. This way, parameters of the neural network are split into statically meaningless

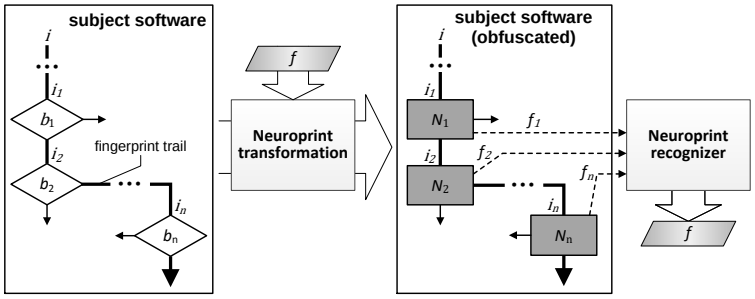


Fig. 6: Integrating fingerprints with the neural-network-based control obfuscating.

components inside instructions that are distributively placed over the software's code body. In addition, the updating of the weight matrix is covered with complicated dynamic data dependencies. Therefore, this makes it hard to determine the resulting networks without actually reach the corresponding obfuscated branches at runtime.

To add complexity in a potential analysis of the weight matrix, we also use the same memory region for weight matrixes of multiple neural networks. After one network finishes its task, the weight matrix is updated into a new one for the next neural network to be used. Updating of the matrix for each network is only completed right before it being queried in the control transfer, and it is overwritten right after the transfer. An attacker could only obtain the correct view of a network when program's execution reaches the corresponding conditional branch.

IV. Neuroprint: Towards Integrated Fingerprinting

Remind in Section I-A we explained that to the existing dynamic fingerprinting schemes, *forming additional program semantics that serve the watermark/fingerprint exclusively* is a principle by default. In this paper, we challenge the necessity of doing so. Although we agree that it is inevitable to cause changes to a program in order to introduce new information, we doubt that such changes have to reach semantic level. Dynamic fingerprinting should imitate the "perceptually insignificant" objective in the design of digital fingerprinting, and construct fingerprints in a *semantically insignificant* way. Towards such direction, we suggest a new notion on dynamic fingerprinting called *integrated fingerprinting*.

Let $\mathbf{F_I} : \{E_I(\cdot), R_I(\cdot)\}$ represent an integrated fingerprinting scheme. Given a program $\mathcal{P} = \langle \Sigma, \imath, \tau \rangle$ and any $x \in \imath$, we use

$$\tau(x) = \{(x, s_0), (s_0, s_1), \cdots, (s_{n-1}, s_n)\} \tag{5}$$

to represent $\mathcal{P}$'s transition sequence when initiated from x (in which $\forall k \in \{0, 1, \cdots, n\}, s_k \in \Sigma$). $\mathbf{F_I}$ then defines a pair of encoding/decoding functions F_{en} and F_{de}.

- F_{en} receives a transition sequence $\tau(x)$ and a message m as input, and outputs a new transition sequence such that $\Theta(F_{en}(\tau(x), m)) = \Theta(\tau(x))$;
- F_{de} consists of a combination of computations that, given a transition sequence of a program, results in a message that satisfies $F_{de}(F_{en}(\tau(x), m)) = m$.

Accordingly, we let $\mathbf{F_I}$'s fingerprint embedding and recognition be described by

$$\begin{cases} \mathcal{P}_f = E_I(\mathcal{P}, \{f, i\}) = \langle \Sigma', \imath, \tau' \rangle \\ f = R_I(\mathcal{P}_f, i) = F_{de}(\tau'(i)) \end{cases} \tag{6}$$

in which, if $\tau_f = F_{en}(\tau(i), f)$, then $\Sigma' = \Sigma \bigcup \tau_f$ and $\tau' = (\tau - \tau(i)) \bigcup \tau_f$. The meaning of such an $\mathbf{F_I}$ is that instead of bringing in new functional modules into the program, the fingerprint f here is turned into a secondary effect of τ' (non-semantic) while ensuring $\Theta(\mathcal{P}_f) = \Theta(\mathcal{P})$. To decode a so embedded fingerprint, $\mathbf{F_I}$'s recognizer R_I has to rely on the protected program's own behaviors. This bounds it with the program such that the difficulty of removing or destroying the fingerprint while keeping the program itself intact can be increased. Furthermore, integrated fingerprinting could also resist collusive attack since given any $f_1 \neq f_2$, $\mathbf{F_I}$ keeps $\Theta(\mathcal{P}_{f_1}) = \Theta(\mathcal{P}_{f_2})$. As a result, the adversary can hardly take advantages of comparing instances of the program that carry different fingerprints.

One may immediately come with a question: *is integrated fingerprinting feasible*? On top of our neural-network-based control obfuscation (see Section III), we further propose an dynamic fingerprinting scheme called *Neuroprint* as the first realization of integrated fingerprinting.

A. Design Overview

A key requirement of integrated fingerprinting is to embed the fingerprints as a secondary characteristic of the execution so that the semantics of the fingerprinted program remain unchanged. Our idea of realizing this is to represent the fingerprints as some kind of runtime memory states (non-output) in program execution. In view of the attractive features provided by neural networks, in particular, the incomprehensibility as discussed in Section II-A, we find that embedding the fingerprints as part of the neural network outputs, could be an attractive solution.

Recall the definition of structural operational semantics mentioned in Section I-A. Assume two programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are semantically equivalent (i.e. $\Theta(\mathcal{P}_1) = \Theta(\mathcal{P}_2)$), it is then necessary that one of their triples $\langle \Sigma(\mathcal{P}_1), \imath(\mathcal{P}_1), \tau(\mathcal{P}_1) \rangle$ and $\langle \Sigma(\mathcal{P}_2), \imath(\mathcal{P}_2), \tau(\mathcal{P}_2) \rangle$ can be reduced to (*but not necessarily be equal with*) the other. In other words, *a program transformation can be semantic-preserving even when it is not state-preserving*. Code obfuscation is by principle a transformation

that aims to express a program's original semantics with a set of much more complex execution states, thus the difficulty of reverse engineering the resulting software instance gets increased. Therefore, if designed properly, the complicated execution states created by code obfuscation might be useful for exhibiting extra information (like software fingerprint) at runtime.

In our neural-network-based obfuscator, the neural network output is used as an offset to calculate the control transfer target address. On the x86 architecture, conditional branching via short jumps has an 8-bit offset to index the control transfer target. Given that the obfuscator adopts integer neural network, this leaves 24 bits unused in its output which is a 32-bit integer when running on a 32-bit architecture. Such unused portion of the neural network output makes it possible to encode dynamic software fingerprints within program's own semantics.

As demonstrated in Figure 6, given a secret input i and a fingerprint f, Neuroprint first runs the subject program with i in order to traverse the corresponding execution trace (i.e. the *fingerprint trail* as mentioned in Section I-C), during which:

- it selects on the fingerprint trail a collection of conditional branches $b_1, b_2, \cdots, b_n$ as candidates to carry f; and
- it records the input values $i_1, i_2, \cdots, i_n$ taken by each $b_k (k = 1, 2, \cdots, n)$ during this specific execution.

After that, Neuroprint divides f into a series of fractions $f_1, f_2, \cdots, f_n$ and replaces branches b_1 to b_n with neural networks N_1, N_2, $\cdots$, N_n. Each $N_k (k = 1, 2, \cdots, n)$ here is specifically constructed such that when input with i_k, it responses with an output containing both the control offset of the destination branch and the assigned fingerprint fraction f_k. Later, the embedded fingerprint can be extracted by using an external recognizer to observe the execution of the transformed software with the input i, during which the fingerprint fractions will be successively presented by the neural networks upon being invoked to perform the conditional control transfers they represent.

Furthermore, Neuroprint also performs obfuscation to the program's other branches using neural networks similar to N_k, except that these networks possess no fingerprint fractions. Such networks then take the role of *decoys*, providing covers for the fingerprinted networks. In fingerprinting different copies of a program, Neuroprint makes sure that each distinct neural network it generates is trained using a different initialization such that any two of them, whether simulating the same branch of the program or not, would not be identical. As the result, diff analyses on the resulting software copies, which drives the collusive attacks, would hit on the construction code of all the neural networks from both type. The large number of decoys mixed in the identified differences, together with the incomprehensibility of neural networks, make it harder to determine the actual location where the fingerprint fractions are deployed. Meanwhile, since the system is built on top of the control flow obfuscator, compromising any of the neural networks is as least equally difficult as if revealing the conditional branch protected by that network.

In the next two subsections, we present details of the fingerprint embedding and recognition processes.

B. Fingerprint Embedding

The design of fingerprint embedding has a large impact on the security of its recognition. In practice, the recognition process of a fingerprinting scheme must assume that integrity of the fingerprinted software has already been compromised in any possible way. For example, adversaries may also insert bogus fingerprint components of their own into the software, hoping that the forgeries would sabotage the actual fingerprint from being correctly recognized. That is why our design of the fingerprint embedding focuses on the integrity of the fingerprint (fractions).

In addition to the control transfer offset and the fingerprint fractions in the output of neural network, we also insert the integrity check. To show how this works, we denote the four distinct bytes of the neural network output as **Byte0** to **Byte3**. Out of these four bytes,

1) **Byte0** is called the *control section* storing offset of the control flow transfer;
2) **Byte1** is the *fingerprint section* containing the embedded fingerprint fraction; and
3) **Byte2** and **Byte3** together make up the *authentication section*, which consist of 2 HMACs **MacSlf** and **MacMu**.

Here **MacSlf** and **MacMu** denote the *self authenticator* and the *mutual authenticator*, respectively. Let $\mathcal{H}(msg, key)$ denote the last 8 bits of the output of a keyed hash function. Given any fingerprint fraction f_k and its trigger input $i_k (k = 1, 2, \cdots, n)$,

$$\begin{cases} \forall k, \mathbf{MacSlf}_k = \mathcal{H}(f_k || i_k, i); \\ \forall k \neq n, \mathbf{MacMu}_k = \mathcal{H}(f_{k+1} || \mathbf{MacSlf}_{k+1}, i); \\ \mathbf{MacMu}_n = \mathcal{H}(f_1 || \mathbf{MacSlf}_1, i). \end{cases} \tag{7}$$

In this way, each fingerprinted output provides authentication not only for itself but also for its successor. Fig. 7 shows the structure of the reformatted neural network output used in Neuroprint system.

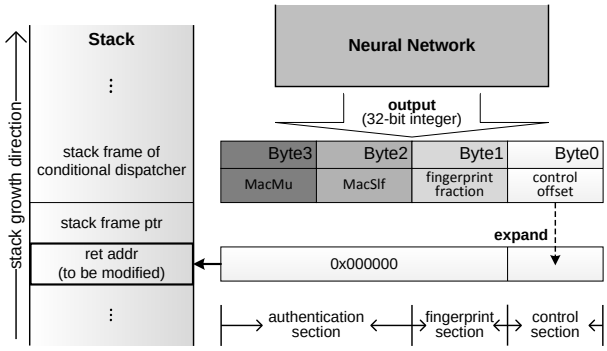


Fig. 7: Output of neural network and its location in memory

Training of the neural network makes sure that its output conforms to the definition above with the correct input i_k. For those inputs that goes to the other branch (and all the inputs of decoy neural networks), we let **Byte3**, **Byte2** and **Byte1** be some pre-determined random numbers. We present the implementation of this design in Section V.

C. Fingerprint Recognition

Besides being able to work correctly on software copies that are potentially sabotaged, the recognizer for a software fingerprinting scheme should also require few code features from the fingerprint modules, especially those that are observable statically; otherwise, the pattern that the recognizer uses could be exploited by the adversaries for malicious purposes. For example, such features could be removed to make the fingerprint unrecognizable. Therefore, we build the recognizer of Neuroprint mainly based on data-level patterns as opposed to code patterns and bound such data with original semantics of the program.

The Neuroprint recognizer works by subjecting the software copy to execute in hardware single-stepping during which the recognizer records the neural network outputs. This requires to first locate the conditional dispatchers (see Section III-A) such that the neural network computation can be put under monitoring. We do this by exploiting the special runtime feature of the dispatcher that it does not return to its call-site (due to obfuscated control flow) when the fingerprint generation is activated.

As discussed in Section IV-B, the recognizer might be dealing with software that has been compromised and modified. Therefore, it is essential that we accept only fingerprint fractions that are authentic. We do this by verifying the authentication section of the neural network output. Let $o_t = \mathbf{Byte3}_{o_t}||\mathbf{Byte2}_{o_t}||\mathbf{Byte1}_{o_t}||\mathbf{Byte0}_{o_t}$ denote one of the neural network outputs collected by the recognizer for an input i to the program and i_t to the corresponding dispatcher. We check that

$$\mathbf{Byte2}_{o_t} = \mathcal{H}(\mathbf{Byte1}_{o_t}||i_t, i) \tag{8}$$

and

$$\mathbf{Byte3}_{o_t} = \mathcal{H}(\mathbf{Byte1}_{o_{t+1}}||\mathbf{Byte2}_{o_{t+1}}, i) \tag{9}$$

before accepting the fingerprint fraction. Assuming the secrecy of i, it is hard for any unauthorized party to distinguish a fingerprint fraction from a random padding. Meanwhile, our recognizer is able to accept only authentic fingerprint fractions and will reliably reconstruct the fingerprint.

Since an authentication protocol is adopted, possibility of false recognition needs to be discussed. For Neuroprint, this happens when the software is not running on the fingerprint trail, yet output of an encountered neural network (carrying no fingerprint fraction) happens to satisfy Equation 8 and 9. Although the two MACs used in our design are each 8-bit long only, we argue that the possibility of a false recognition (as mentioned above) under such setting is still trivial. This is because that a successful recognition would eventually require a whole chain of mutual authentication be satisfied, which is unlike to accidentally happen on an unexpected execution trail.

V. Implementation

Our implementation of Neuroprint consists of mainly a software analyzer, a neural network trainer, and a source code analyzer/rewriter. The prove-of-concept implementation is of fewer than 1000 LOC in total. The system takes as input the source code of a program in C/C++, an input i, and the fingerprint f, and outputs the modified source code which can then be compiled into the obfuscated and fingerprinted version. We start with analyzing the original subject program's source for the purposes of identifying and selecting conditional branches for obfuscating and fingerprinting. We then insert place holders to the source for neural network construction and the dispatchers. The modified source is then compiled into binary executable for more comprehensive static and dynamic analysis to calculate control offsets. After that, we train the neural networks and insert code into the place holders correspondingly.

A. Source Code Analysis and Preliminary Rewriting

Recall that the dispatcher to be inserted into the obfuscated (and fingerprinted) version makes control transfers by adding an offset to the return address. We therefore need to find out such offsets before source code rewriting starts. However, these offsets may change during re-compilation after the source is modified. To handle this difficulty, we apply a two-phase source code rewriting on the subject program (similar approach is used in the original work of control flow obfuscator [1]).

In the first phase, we select branches to be obfuscated directly from the software's source code and then perform a preliminary rewriting to insert place holders for the source code of neural network construction and the dispatcher. The place holders contain actual code for Neuroprint's corresponding functionality, e.g. instructions of the conditional dispatcher function (see Section III-A) and the weight matrix updating code (see Section III-C). Except that operands in these instructions are not yet of their correct values because at this phase they are still underdetermined. After this rewriting, the modified program source is compiled and the resulting binary is statically analyzed to calculate the correct offset of branches identified for obfuscating. We also perform dynamic analysis to obtain the fingerprint trail and the corresponding inputs to the conditional branches selected, see Section IV-A.

B. Neural Network Training

Training of the neural network resembles that in the control flow obfuscator [1]. Recall that the goal is to let the networks simulate software's branching behaviors as if they are classification tasks. Without loss of generality, program's conditional logic, let $i \in \mathbb{I}$ be its input variable, can be described as

$$l(i) = \begin{cases} 1, i \in \mathbb{A} \subset \mathbb{I} \\ 0, i \in \mathbb{B} = \mathbb{I} - \mathbb{A} \end{cases}. \tag{10}$$

Thus correspondingly, neural network used to replace such logic should behave as

$$N(i) = \begin{cases} v_1, i \in \mathbb{A} \subset \mathbb{I} \\ v_2, i \in \mathbb{B} = \mathbb{I} - \mathbb{A} \end{cases} \tag{11}$$

in which the detailed format of v_1 and v_2 is as described in Section IV-B. Training such a network requires to establish a set of input-output samples that consists of 2 groups: one

contains samples with input values selected from $\mathbb{A}$ and target output set to v_1; while those of the other groups have input from $\mathbb{B}$ and v_2 as target. After that, given a pre-defined network topology, our training process randomly initiates a set of network parameters, then apply the learning algorithm to adjust the parameters until the resulting network fits the behavior of the given training set.

Since for different input values from the same branch, our neural network is meant to respond with the exact same output, function $N(\cdot)$ in Equation 11 to be simulated by the network is in fact quite simple. Therefore, thanks to neural network's powerful generalization ability, it is not necessary to exhaustively pick all possible values from the general input space $\mathbb{I}$ to build the training set. In fact, our experience shows that when training samples are discretely selected from $\mathbb{A}$ and $\mathbb{B}$, simulation errors occurred in the resulting neural networks (if any) appears only on input values that are close enough to the "boundaries" between the two subspaces. Therefore, it is feasible to let the training set focus only on this error-prone area, and let the generalization effect take care of the rest of the input space. Specifically, our training set includes the same number of (possibly repeated) samples from each input group with a higher density at close vicinity (± 1000) of the boundary values and a lower density elsewhere.

We apply practical swarm optimization (PSO), a sophisticated algorithm widely applied in neural network training [32], [33], as the network learning algorithm. The effectiveness of PSO does not rely on the gradient of target distributions to be simulated; therefore networks trained with it show better accuracy even in the boundary regions. We construct our neural networks to have one input and one output with at most two hidden layers and no fewer than 7 nodes in each of them. We also design a neural network verifier to confirm the correctness of the networks trained. Any flawed neural network that does not always behave correctly over the entire input space will not be accepted, and the training will restart until an accurate network is produced. Different to training protocols for other application scenarios, in obfuscation we already know that for our network, there exists no unexpected input value that does not belong to $\mathbb{I}$. Consequently, so long as the selected training set is representative enough, it is preferred for our training process to produce networks that are over-fitting.

C. Filling Source Code Place Holders

After the neural networks are trained, operands in the place holders of the neural network updating instructions can be refreshed to their actual values, so that now they can correctly construct the neural network weight matrixes upon executed. Since only the value of certain operands are changed in this step, the resulting binary executable has only minimal changes to its instructions, which have no affect on their corresponding addresses. The multi-layer structure of neural networks also makes it easy to build the conditional dispatcher into nested subroutines. For example, the dispatcher could first invoke a subroutine to calculate the network's hidden layer, which would further invoke another subroutine for calculating the output layer. This output layer calculator then uses the network's output to modify the dispatcher's return address that is not in its own stack frame. This way avoids straightforward return address tampering that may be easily noticed, as suggested in [10].

VI. Effectiveness Analysis

As described in Section I-A and I-B, in general a software fingerprinting scheme considers attacks of the following styles:

- *Additive attacks*, which inject into the program bogus components that may be mistakenly recognized as (part of) the fingerprint, to prohibit the original one from being correctly recovered;
- *Subtractive/distortive attacks*, which attempt to remove or corrupt the fingerprint while preserving (most of) the program's original functionality; and
- *Collusive attacks*, which aim to locate the hidden fingerprint by comparing different copies of the program.

These attacks respectively target different aspects of software fingerprinting: collusive attacks for exposing the fingerprint's location, additive attacks for confusing the recognition process with bogus information, and subtractive/distortive attacks for actually compromising the embedded message.

Nonetheless, when it comes to our Neuroprint system in specific, not all the above attack styles could work effectively. Due to our authentication protocol (see Section IV-C), it is next to impossible for an adversary to forge bogus execution states that can disrupt Neuroprint's recognition process. This means that additive attack can hardly become a threat. Our analyses and evaluations therefore focus on the effectiveness of our scheme against subtractive/distortive attacks and collusive attacks.

A. Resilience to Subtractive/Distortive Attacks [1]

Given that our approach embeds fingerprint fractions inside neural networks that obfuscate program's conditional branches, attempts to remove or corrupt the fingerprint yields to either de-obfuscate branches replaced by the networks, or change the networks' behaviors by tweaking their parameters. However, even if a tweaking on the neural networks does cause their output be modified, changes would occur on the lower bits of such output first, then affect the higher bits if the modification is significant enough. Therefore, due to the output format of our neural networks (see Section IV-B), such change would certainly compromise the control section (and thus jeopardize the software's integrity) before even reaching the fingerprint section. As the result, our evaluation focus on de-obfuscation as the major approach of launching a subtractive/distortive attacks against Neuroprint.

We assume that our approach faces the typical MATE adversary who is in full control of the system in which the protected software executes. The objective of the attack, specifically to the obfuscation part of our approach, is to reveal the condition of the control flow that has been simulated using neural networks. If the adversary succeeds, he is then able to either recover the transformed conditional branches into their

TABLE I: Experiment settings for evaluating Neuroprint's resilience against concolic testing

"Condition(Var)"s for testing			applied network topology			
"Var="	"Var>"	"Var≤"	# of inputs	# of hidden nodes		# of outputs
				1st layer	2nd layer	
16, 29	16	29	1	10	-	1
6, 11	6	11	1	15	-	1
4, 20	4	20	1	8	8	1
2, 13	2	13	1	12	12	1

original form, or replace the neural networks while keeping the program semantics intact.

We focus on the worst possible scenario for the protected program, in which the adversary applies latest state-of-the-art dynamic analysis tools with symbolic/concolic testing in the attack [5]–[8]. These are the latest dynamic analysis tools with the purpose of reverse engineering all control flows of a subject program. The typical analysis starts by executing the subject program with a random input and obtaining the corresponding trace information, i.e., branches and the corresponding conditions. It then uses the information collected to deduce new inputs (with the help of symbolic testing and usually a theorem prover) that may trigger different execution traces (branching) until all possibilities are exhausted.

As already discussed in [1], because of the difficulty of extracting rules from neural networks [17]–[19], revealing conditions of our obfuscated control transfers with concolic testing is computationally impractical. As shown in Fig. 8, each layer of the neural network involves neurons receiving inputs from (every) neuron of the previous layer. Reversing such logic leads to an undetermined linear formula of which the number of potential solutions is extremely large. Chaining such effect to multiple layers quickly enters into a combination explosion that is beyond the computation capability of any existing solver, which consequently stops concolic testing from functioning.

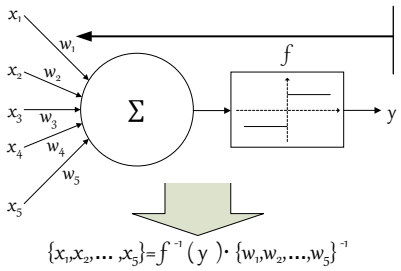


Fig. 8: Reversing the logic of one neuron in a neural network

To further evaluate the effectiveness of our scheme against concolic testing, we performed a simulation in which an extremely simple program

```
void main(){
  int Var=SomeValue;
  if(Condition(Var)) Var++;
}
```

is transformed using our approach and tested with concolic testing tools afterward. We use such a subject program so that the evaluation can exclusively focus on the effectiveness of applying neural network in control obfuscation (given that the program does not have other components except the conditional logic to be obfuscated). For the same purpose, we do not apply indirect control transferring to the subject program in this evaluation to rule out the known drawbacks of concolic testing while handling indirect branches [34]. Neural networks used in the experiments are given a series of different topologies, and the branch `if(Condition(Var))` in the program is given a series of equal/unequal branch conditions as shown in Table I.

We selected two popular analysis tools for the experiment, KLEE [8] and TEMU of the Bitblaze platform [5]. KLEE is chosen because it is a representative execution-generated testing approach which provides powerful path exploration by mixing concrete and symbolic execution [35], and is also well maintained by an active community of contributors from both academia and industry [36]. We use KLEE to test the effect of our design in impeding analysis that aims to probe unexecuted paths of the program to determine their trigger conditions. TEMU, on the other hand, works directly on binary executable and performs in-depth concolic testing for execution path verification. To our best knowledge, this is currently the only open source platform providing symbolic execution assisted by dynamic taint analysis. When TEMU traces an execution path of the program, it generates a constraint set in which symbolic values are marked as tainted, then feeds it to the constraint solver. The solver then solves for a test case that is supposed to trigger the given execution path. Although TEMU doesn't actually do path exploration, bringing it into our evaluation still provides a convincing demonstration on how our design works against concolic testing.

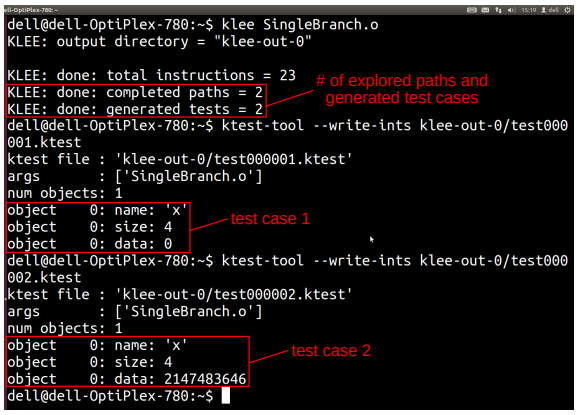
1) Evaluation results with KLEE: As in Figure 9, analysis result shows that while KLEE can easily explore both paths of the original program (before obfuscation) and generate test cases for them correspondingly, it can only detect a single feasible path on the program obfuscated. This indicates that our obfuscation successfully hinders concolic testing from understanding the programs' control structures.

Unfortunately we can only go this far since KLEE provides no more information (e.g., errors occurred or unexpected situations happened) to assist the user other than its final analysis result[1]. According to the documentations [8], we believe that KLEE reaches the branch obfuscated and is unable to determine whether both branches are reachable because its constraint solver fails to find a different output from the neural network.

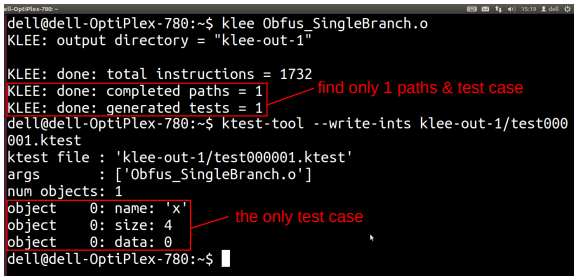
[1]The output of KLEE includes only the number of paths it discovered along with one test case for each path.

TABLE II: Result of TEMU's execution verification on binary executable of the test cases.

statement (unequal)		> 16	> 6	> 4	> 2	≤ 29	≤ 11	≤ 20	≤ 13
input value		-16							
verification result	original	-16							
	obfuscated	NA							
# of constraints	original	733							
	obfuscated	13751	18759	28931	31584	12809	18759	28931	31658
statement (equal)		$= 16$	$= 6$	$= 4$	$= 2$	$= 29$	$= 11$	$= 20$	$= 13$
input value		16	6	4	2	29	11	20	13
verification result	original	16	6	4	2	29	11	20	13
	obfuscated	NA							
# of constraints	original	2566							
	obfuscated	12809	18759	28931	31584	12809	18759	28931	31584



(a) On the original program



(b) On the obfuscated program

Fig. 9: Result of KLEE's path exploration.

2) Evaluation results with TEMU: The experiment on TEMU is more complex compare to that on KLEE. As shown in Table II, 16 different settings of the conditions are given to the example program, 8 of which are for inequality conditions and the rest are for equality conditions. TEMU is then used to perform execution verification on all 16 settings of the program as well as their obfuscated versions, of which the results are compared.

Good thing is that the above simulations show positive results in consistence with the assumption we made. From Table II we can see that for the original program with all 16 branch conditions, TEMU is able to precisely return the input values that cause the execution paths it observes, indicating successful verification. For all obfuscated programs, however, the constraint expressions TEMU generated for the dynamic tainted traces expanded significantly, and it fails to return a valid input value. This result provides more solid evidence indicating that our method managed to make the protected conditional behaviors too complicated for the constraint solvers to reason about.

It should again be emphasized that these evaluations are taken on programs consisting of only *a single obfuscated control structure*. It certainly infers that the complexity of obfuscated control flow would be way beyond existing analysis tools' capability, should our method be applied on actual applications.

B. Resilience to Collusive Attacks

To begin with, it is necessary to emphasize again that for existing dynamic fingerprinting designs, the advantage provide to the adversary by collusive attacks is *a simple yet effective way of locating the embedded fingerprint*. Compromising the fingerprint after its disclosure is not in the scope of this type of attacks (it is still the job for subtractive/distortive attacks).

First we demonstrate a collusive attack launched against conventional dynamic fingerprinting approaches. Specifically, we apply the dynamic path-based watermarking proposed in [10] (which is directly transplanted to work as fingerprinting as said in Section I-A) on a benchmarking program `bzip` picked from the SPEC2006 suites, and embed a message of 1-byte long inside its function `compress`. Two instances of the program are embedded with individualized message `0x7A` and `0x5F`. After that, their corresponding binary executables (in which the entry of `compress` is at `0x004074C8`, while the location of fingerprint payload is from `0x00407508` to `0x00407569`) are analyzed using `IDApro`, a well-known binary analysis tool, with the purpose to mark out any static-level differences between their code texts. As shown in Figure 10, in the diff analysis `IDApro` successfully located 3 different control transfers within the region of the payload code, which corresponds to the 3 different bits between the embedded fingerprints (of which binary form are respectively `01`**`0`**`11`**`1`**`11` and `01`**`1`**`11`**`0`**`10`). It is therefore not hard to infer that with enough instances of the program containing different fingerprints, instructions representing all 8 embedded bits would eventually be located, at which point the fingerprint module is completely exposed. It is worth emphasizing that although only the path-based watermarking is tested here, the same result will also be observed on other conventional dynamic watermarking/fingerprinting approaches, e.g. [11], [12] and [37]. In this subsection, however, we show that when

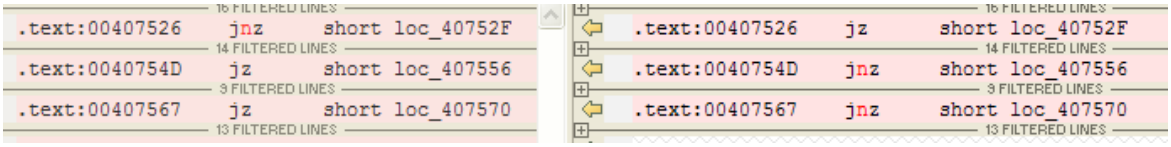


Fig. 10: A demonstration on the effect of collusive attack against path-based fingerprinting.

launched against Neuroprint, collusive attacks can hardly bring the same advantages to the adversary.

Again, we assume that the adversary we face is in full control of the protected software, expect the input i for triggering the fingerprint remains secret to him. Given that collusive attacks work via diff analyses among program copies containing distinct fingerprints, we also make a conservative assumption that Neuroprint always selects the same set of branches for fingerprinting and decoy in each copy of the program, only every neural network it deploys is trained with a distinct initialization as described in Section IV-A. Meanwhile, the adversary is considered to have somehow overcome the aliasing created during the dynamic construction of the neural networks and had the parameters of each of them determined.

The above assumptions set up a scenario in favor of the adversary, in which when 2 copies of the fingerprinted program are compared in a collusive attack, instructions related to all neural networks in both copies will be found out and grouped in pairs according to their position in the program. For the adversary, the task is simply to make a decision for each pair of neural networks, to determine if

1) the two networks are functionally identical, i.e., they have no fingerprint fraction embedded? Or
2) the networks are different, i.e., they are fingerprinted?

We show that even in this condition, the adversary is still not able to reliably distinguish the fingerprinted neural networks from the decoy ones.

In Section IV-B we already described the self and mutual authentication of Neuroprint's fingerprint formatting. Satisfying any of the two protocols requires both the secret input i for the program and each fingerprinted neural network's input during the execution triggered by i. And the fingerprint is recognized only when a complete mutual authentication chain checks out while each neural network output on the chain passes self authentication (see Section IV-C). Since the adversary here is considered to have no knowledge about the secret input i, investigating the input-output behavior of the neural networks of our approach is not a feasible way for him to find out the networks carrying fingerprint fractions with high confidence. When performing the authentications on an incorrect execution trail, the search would either find no suspicious neural networks or come back with false candidates.

What's left for the adversary is to see whether comparing parameters of the two neural networks in a pair may show anything that helps distinguishing their functional differences. However, as explained in Section II-B, the random initialization to neural network typically result in completely different sets of parameters even when representing the same input-output behavior. This makes it impossible to differentiate neural networks all by their parameters. In order to verify this argument, we assume a branch controlling a 37-byte-long code block (i.e., the offset to skip the branch is `0x25`) with `if (x==10)` be its condition. Without loss of generality, we also assume that there is a fingerprint fraction of `0x5F` and the other three bytes of the neural network output being `0xE70891` and `0xE708FF` for the two branches, respectively. We then train two groups of neural networks, with

- $\mathbb{N}_a$ representing neural networks trained only to obfuscate the branch with output being `0xE7089100` when input is 10, or `0xE708FF25` otherwise;
- $\mathbb{N}_b$ representing neural networks that carry the fingerprint fraction when input is 10, i.e., output being `0xE7085F00` when the branch is taken, or else they still output `0xE708FF25`.

The topology of all neural networks here consists of 1 input and output respectively, as well as 2 hidden layers with 12 fully connected nodes in each (see the last row of Table I). As we can see, neural networks in $\mathbb{N}_a$ and $\mathbb{N}_b$ are different only on a single input-output pairing. We then use a set of initialization $\mathbb{I}$, which contains a total of 30 different settings, to train the networks of both groups. Two networks $netA_k \in \mathbb{N}_a$ and $netB_k \in \mathbb{N}_b$ are a pair resulting from the same initialization. Finally, we merge the two groups of networks together $\mathbb{N} = \mathbb{N}_a \bigcup \mathbb{N}_b$ and apply a total of six well-known clustering algorithms (provided in scikit-learn, an open source data mining and analysis toolkit [38], see Table III) to see whether $\mathbb{N}_a$ and $\mathbb{N}_b$ can be correctly separated.

TABLE III: Clustering algorithms

Algorithm	# of clusters specified?	Use case
K-Means	√	Even cluster size, flat geometry, not too many clusters
Mean-shift	×	Many clusters, uneven cluster size, non-flat geometry
AC[1]	√	Many clusters, possibly connectivity constraints, non Euclidean distances
DBSCAN	×	Non-flat geometry, uneven cluster sizes
SC[2]	√	Few clusters, even cluster size, non-flat geometry
AP[3]	×	Many clusters, uneven cluster size, non-flat geometry

[1] Stands for Agglomerative Clustering.
[2] Stands for Spectral Clustering.
[3] Stands for Affinity Propagation.

Figure 11 shows the result of the clustering. It appears that K-Means, Agglomerative Clustering, and DBSCAN are completely out in the analysis. Spectral Clustering labeled the networks into 2 groups with significant errors. Both Mean-shift

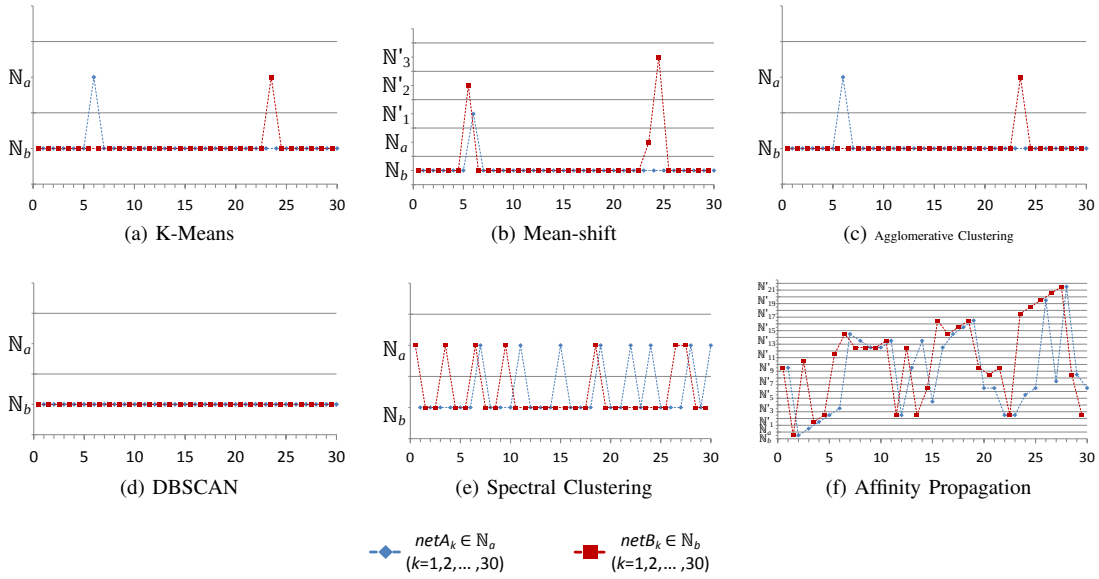


Fig. 11: Results of clustering neural networks of $\mathbb{N}_a \bigcup \mathbb{N}_b$ using the listed algorithms

and Affinity Propagation ended up creating multiple clusters that do not exist. Note that unlike K-Means and Agglomerative Clustering, DBSCAN doesn't require the number of clusters to be specified in advance. This shows that all networks from both $\mathbb{N}_a$ and $\mathbb{N}_b$ are indistinguishable by clustering their parameters.

Finally, it is worth mention that the only factor which determines how a neural network works is its topology. This means that given two neural networks that are built in the same structure, serve for the same branch, then the computing procedure of them are exactly the same even if their outputs contain different random padding or fingerprint. Put the above in summary, the design of Neuroprint makes it hard to discriminate its fingerprinted and decoy neural networks at either semantic level or instruction level. Therefore, collusive attacks on Neuroprint provides no extra advantage to the the adversary, considering a comprehensive de-obfuscation (as in a pure subtractive/distortive attack) on the protected program is still the best possible case for him.

C. Performance Overhead

To evaluate the potential overhead that Neuroprint may add to the software, we apply our technique on a group of benchmarking programs selected from the SPEC2006 suites and compare the performance of their Neuroprinted version with that of the original ones. We still use the same 1-12-12-1 topology for the neural networks (as was in Section VI-B), and the system is instructed to transform as many branches as possible in an attempt to simulate the worst possible performance cost. Table IV summarizes the average time it takes for the programs to execute and the corresponding overhead across 50 runs.

Results show that the overhead of Neuroprint ranges from 1.6% to 6.5% while transforming between 100 and 2200 conditional branches, which is small. Also note that our approach requires constant additional runtime memory space for maintaining the neural network parameters since multiple networks share the same memory units.

TABLE IV: Overhead on SPEC benchmark programs

Program	# of branches	Performance		
		Original	Transformed	Overhead
bzip2	155	103.59ms	105.46ms	1.805%
hmmer	500	1425.14ms	1448.66ms	1.650%
lbm	102	82516.22ms	84939.60ms	2.937%
mcf	135	4431.11ms	4563.36ms	2.985%
sjeng	1738	6377.40ms	6755.29ms	5.925%
soplex	2148	102.80ms	109.46ms	6.479%

D. Limitation

Neuroprint is capable of defeating sophisticate automated reverse engineering techniques. However, the mechanism of obfuscation determines that any obfuscated control transfer can be treated as a black-box and brutally tested for all possible inputs. And within all kinds of conditional branches, those with inequality conditions are particularly vulnerable because intrinsically they can be explored using binary search. Thus for control transfers of such type, Neuroprint is secure only against automated analyses, but can be circumvented using side-channel attacks, like the black-box testing mentioned here. That said, we emphasize that the branches with equality condition do not share the same vulnerability against black-box testing. In such a control transfer, only a single input triggers one of the execution pathes, which cannot be searched heuristically. Therefore, by choosing equality branches as fingerprint candidates, Neuroprint can avoid from letting its effectiveness be undermined by the mentioned side-channel attacks.

Furthermore, it is also worth mention that as a potent control obfuscator, Neuroprint may also use its neural networks to

construct *opaque predicates* which appears like conditional branches but in fact always directs execution to the same position [39]. Determining whether a neural network represents an opaque predicate or an actual conditional branch also requires to exhaustively test all possible inputs of it. Therefore, selectively inserting such opaque predicates into subject program, and consider them as available candidates to carry fingerprints, would also increase the strength of Neuroprint.

VII. Related Works

A. Software Watermarking and Fingerprinting

Despite their differences on application scenarios, the border between software fingerprinting and watermarking has always been quite vague (basically only on the usage of the embedded secret signatures). Therefore, most of the previous works only discusses watermarking since it seems the same techniques can be easily extended to fingerprinting.

There has been extensive work on both static [40], [41] and dynamic [10]–[14], [37], [42], [43] watermarking solutions. The dynamic approaches mainly fall into two categories. Graph-based approaches, first introduced by Collberg et al. [44] and currently the most well-understood technique, encode the secret signatures into heap-allocated graph structures such that they could be picked up by the recognizer via garbage collection [11], [12], [42]. In other dynamic approaches, signatures to be embedded are turned into specialized execution states, e.g. multi-thread behavior [37], conditional branches [10], [13], or opaque predicate values [43]. Existing work has not consider seriously the uniqueness of software fingerprinting (especially that of dynamic fingerprinting).

All the above designs insert to the subject program their special components (e.g., data structures representing graph nodes [11], [12] and special thread components [37]) as well as related code blocks that make no valid contribution to the software's original functionality. In the fingerprinting scenario, this significantly compromises their effectiveness since the fingerprint-specific executions would inevitably bring distinguishable semantic-level differences to each fingerprinted software copy.

B. Control Flow Obfuscation

Control flow obfuscation is one of the major methods of code obfuscation. It aims to make the control flow of a given program difficult to understand [45]. Sharif et al. presented a conditional code obfuscation scheme that uses hash function and self-decrypting code to protect equal branch conditions and the modules they control [16]. There were also attempts to turn control transfers into signals (traps), and then introduce dummy transfers and junk code to confuse static analysis [46]. Others exploit the limitation of symbolic execution in solving unrolling loops [27].

Recently, a design that makes code obfuscation a constituent part of software watermarking, namely the *Droidmarking* system [14], was proposed. Exploiting self-decrypting code as originally used in [16], Droidmarking establishes interdependence between its watermark payloads and the carrier software, and achieves satisfactory resilience against evading attacks despite causing non-stealthy modifications. However, the non-stealthy mode is not applicable to software fingerprinting since the embedded fingerprints are meant to be only recognizable by authorized parties instead of making it available to the public.

Our Neuroprint system differs from all existing dynamic techniques in that it makes the fingerprint a built-in portion of the software's original semantics via control obfuscation. Therefore, the resilience against attacks targeting weaknesses of traditional dynamic fingerprinting is significantly improved.

VIII. Conclusion

We proposed a novel mode of dynamic software fingerprinting, namely the integrated fingerprinting mode, which encodes fingerprint messages as a secondary effect of a special semantic-preserving transformation on the subject program. Furthermore, We presented a composite control obfuscation/-dynamic fingerprinting approach named Neuroprint, as the first realization of the integrated fingerprinting system. Evaluations demonstrated that with the strong security properties of the underlying neural-network-based obfuscator and the specifically designed fingerprint authentication mechanism, Neuroprint achieves satisfactory resilience against various types of attacks. Simulation on the SPEC benchmarks also indicated that Neuroprint is efficient and comes with acceptable memory cost.

We believe that a fresh and interesting view could be opened by this work, indicating that it is possible to integrate software fingerprints into semantic-preserving transformations like program obfuscators.

References

[1] H. Ma, X. Ma, W. Liu, Z. Huang, D. Gao, and C. Jia, "Control flow obfuscation using neural network to fight concolic testing," in *Proceedings of the 10th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2014.

[2] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam, "Using symbolic execution to guide test generation," *Softw. Test. Verif. Rel.*, vol. 15, no. 1, pp. 41–61, 2005.

[3] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P)*, 2007, pp. 231–245.

[4] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest editors' introduction: Software protection," *IEEE Softw.*, vol. 28, no. 2, pp. 24–27, 2011.

[5] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security (ICISS). Keynote invited paper*, 2008.

[6] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, 2006, pp. 419–423.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 10:1–10:38, 2008.

[8] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[9] K. Fukushima and K. Sakurai, "A software fingerprinting scheme for java using classfiles obfuscation," in *Proceedings of Information Security Applications: 4th International Workshop (WISA 2003)*, ser. LNCS, K.-J. Chae and M. Yung, Eds., vol. 2908. Springer Berlin Heidelberg, 2004, pp. 303–316.

[10] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI)*, 2004, pp. 107–118.

[11] C. Collberg, C. Thomborson, and G. M. Townsend, "Dynamic graph-based software fingerprinting," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 6, p. 35, 2007.

[12] W. Zhou, X. Zhang, and X. Jiang, "Appink: Watermarking android apps for repackaging deterrence," in *Proceedings of the 8th ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, 2013, pp. 1–12.

[13] G. Myles and H. Jin, "Self-validating branch-based software watermarking," in *Proceedings of the 7th International Workshop of Information Hiding (IH)*, 2005, pp. 342–356.

[14] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding android application repackaging," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*, 2014, pp. 635–646.

[15] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *Proceedings of the 16th Annual Conference of Computer Security Applications (ACSAC)*, 2000, pp. 308–316.

[16] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2008, pp. 321–333.

[17] A. B. Tickle, R. Andrews, M. Golea, and J. Diederich, "The truth will come to light: directions and challenges in extracting the knowledge embedded within trained artificial neural networks," *IEEE Trans. Neural Netw.*, vol. 9, no. 6, pp. 1057–1068, 1998.

[18] M. Golea, "On the complexity of rule extraction from neural networks and network querying," in *Proceedings of the Rule Extraction From Trained Artificial Neural Networks Workshop, Society For the Study of Artificial Intelligence and Simulation of Behavior Workshop Series (AISB)*, 1996, pp. 51–59.

[19] G. G. Towell and J. W. Shavlik, "The extraction of refined rules from knowledge based neural networks," *Mach. Learn.*, vol. 13, no. 1, pp. 71–101, 1993.

[20] G. Cybenko, "Approximations by superpositions of sigmoidal functions," *Math. Control Signals Syst.*, vol. 2, no. 4, pp. 303–314, 1989.

[21] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[22] Y. Bengio, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[23] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[24] J. Chorowski and J. M. Zurada, "Extracting rules from neural networks as decision diagrams," *IEEE Transactions on Neural Networks*, vol. 22, no. 12, pp. 2435–2446, 2011.

[25] D. Bhalla, R. K. Bansal, and H. O. Gupta, "Function analysis based rule extraction from artificial neural networks for transformer incipient fault diagnosis," *International Journal of Electrical Power & Energy Systems*, vol. 43, no. 1, pp. 1196–1203, 2012.

[26] R. Setiono, B. Baesens, and C. Mues, "Rule extraction from minimal neural networks for credit card screening," *International journal of neural systems*, vol. 21, no. 4, pp. 265–276, 2011.

[27] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, 2011, pp. 210–226.

[28] C. Johansson and A. Lansner, "Implementing plastic weights in neural networks using low precision arithmetic," *Neurocomputing*, vol. 72, no. 4–6, pp. 968–972, 2009.

[29] C. Tang and H. K. Kwan, "Multilayer feedforward neural networks with single powers-of-two weights," *IEEE Trans. Signal Processing*, vol. 41, no. 8, pp. 2724–2727, 1993.

[30] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[31] R. Ghiya and L. J. Hendren, "Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1996, pp. 1–15.

[32] R. C. Eberhart and Y. Shi, "Particle swarm optimization: Developments, applications and resources," in *Proceedings of the 2001 Congress on Evolutionary Computation (CEC)*, 2001, pp. 81–86.

[33] J.-R. Zhang, J. Zhang, T.-M. Lok, and M. R. Lyu, "A hybrid particle swarm optimization–back-propagation algorithm for feedforward neural network training," *Applied Mathematics and Computation*, vol. 185, no. 2, pp. 1026–1037, 2007.

[34] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proceedings of 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 117–126.

[35] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[36] E. F. Rizzi, "Discovery over application: A case study of misaligned incentives in software engineering," Master's thesis, University of Nebraska, Lincoln, 2015.

[37] J. Nagra and C. Thomborson, "Threading software watermarks," in *Proceedings of the 6th International Workshop of Information Hiding (IH)*, 2004, pp. 208–223.

[38] "scikit-learn," http://scikit-learn.org/dev/index.html.

[39] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[40] G. Myles and C. Collberg, "Software watermarking through register allocation: Implementation, analysis, and attacks," in *Proceedings of the 6th International Conference of Information Security and Cryptology (ICISC)*, 2003, pp. 274–293.

[41] P. Cousot and R. Cousot, "An abstract interpretation-based framework for software watermarking," in *Proceedings of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004, pp. 173–185.

[42] R. Venkatesan, V. Vazirani, and S.Sinha, "A graph theoretic approach to software watermarking," in *Proceedings of the 4th International Workshop of Information Hiding (IH)*, 2001, pp. 157–168.

[43] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[44] C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 311–324.

[45] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscation transformations," Department of Computer Science, The University of Auckland, Tech. Rep. 148, 1997.

[46] I. Popov, S. Debray, and G. Andrews, "Binary obfuscation using signals," in *Proceedings of the 16th conference on USENIX security symposium (USENIX Security)*, 2007, pp. 321–333.