Obfuscating Branch Decisions based on Encrypted Data using MISR and Hash DIGests

Nektarios Georgios Tsoutsos* and Michail Maniatakos‡

*Computer Science and Engineering, New York University, nektarios.tsoutsos@nyu.edu
‡Electrical and Computer Engineering, New York University Abu Dhabi, michail.maniatakos@nyu.edu

Abstract—In this work, we present a novel obfuscation technique that addresses the problem of information leakage in programs with branch decisions based on encrypted data. Our observation is that we can obfuscate individual branch decisions by non-deterministically evaluating all possible execution paths, before “lazily” resolving a digest of the intended branch sequence. We are exploring the effectiveness of signature- and hash-based message digests that are hard to invert and disconnect the branch decisions and the controlling values. Our approach is showcased using a single instruction abstract machine, which is Turing complete and supports branching over encrypted data.

I. INTRODUCTION

An inherent property of general-purpose computation is the ability to make branch decisions at runtime [1]. Typically, branching programs use external inputs or previously generated (intermediate) values to determine the flow of execution [2]. When the computation domain is unencrypted, such decisions reduce to simple arithmetic comparisons between plaintext values. For instance, in CISC processors like the x86, there exist cmp and jne instructions [3] that can be used to implement conditional branching. Similarly, in less complicated single instruction processors (e.g., addleq [4]), such branch decisions are inherent in the particular instruction and depend on a simple comparison with a zero value.

On the other hand, when the computation domain is encrypted (e.g., [5], [6]), program inputs and intermediate data are not in the clear during execution. In such scenarios, any general-purpose program that goes beyond simple function evaluation would have to make branch decisions relying on encrypted data. The underlying problem is that making decisions can leak information about the controlling parameters by observing the outcome. This leakage is essentially an unintended side-channel [7]. Thus, making direct comparisons against known values (e.g., zero) should be avoided, as this could break the security of the encrypted domain. Indeed, comparing an encrypted value with any chosen value allows for trivial binary search cryptanalysis, which could ultimately reveal the encrypted value after a finite number of steps.

To address the limitations of making runtime branch decisions in the encrypted domain, in this work we revisit the way branches are resolved by underlying execution engines (e.g., processor hardware or virtual machine). Our proposed approach employs hash tables and message digest functions [8] over sequences of branch control values, to obfuscate the connection between these values and runtime decisions; in effect, this increases the complexity of recovering information about encrypted values. It should be noted that, as with any obfuscation technique [9], our approach follows a specific threat model, and does not consider unrestricted adversaries that may reverse the applied obfuscation using unbounded resources. Nevertheless, in practice, our proposed use of one-way (hash) functions provides a trade-off between security and area/time overhead for general-purpose applications.

Our contribution in this work can be summarized as follows:

(a) We propose a methodology to obfuscate encrypted control values used for branch decisions, by non-deterministically executing all possible branch paths and “lazily” resolving the deterministic execution trace on the control flow graph.

(b) We employ Multiple Input Signature Register-based message digests as well as standard hashing techniques over sequences of control values, in order to disconnect branch decision parameters from decision outcomes and prevent leaking information about encrypted data.

(c) We reconcile all non-deterministic execution paths as soon as branches are resolved deterministically, ensuring that the plausible program states do not grow exponentially.

The rest of the paper is organized as follows: in Section II we present necessary background details on obfuscation and message digests, while in Section III we elaborate on our assumed Threat Model. In Section IV we describe the proposed obfuscation methodology and discuss our two alternative digest strategies. Our implementation evaluation is discussed in Section V, while related work is presented in Section VI and our concluding remarks in Section VII.

II. PRELIMINARIES

Before describing the details of our contribution, we provide relevant background information on message digests and hashing, which we leverage to obfuscate branch control sequences. At a high level, message digests are functions that receive an input of arbitrary length and are able to generate an output of fixed length [10]. One popular approach to implement such message digests is to use a compression function (e.g., the Davies–Meyer construction) of a given digest size (e.g., 128 bits) and utilize the Merkle–Damgård transformation to chain these digests together (Fig. 1) [11]. Since the message digest size is constant but the message size is arbitrary, due to the pigeonhole principle it is possible that two different input messages generate the same digest (i.e., there is aliasing or collisions) [12]. Since aliasing is generally unwanted,
its probability can be reduced by employing a compression function with a larger digest size.

In VLSI testing, a popular approach that produces message digests very efficiently, is to use linear feedback shift registers (LFSRs) with XORed inputs to each register of the LFSR. This construction, dubbed multiple input signature register (MISR) [13], allows a sequence of messages to be linearly combined with the LFSR states (i.e., superposition principle), and generate a pseudorandom digest. This pseudo-randomness, combined with the ability to generate exhaustive patterns, makes MISRs very attractive primitives in the field of VLSI testing. Similar to LFSRs, MISRs are mathematically modelled using polynomial fields, and each MISR is defined by a characteristic irreducible polynomial (Fig. 2).

One important application of message digests is efficient tagging of data through hash tables. Such tables are organized similarly to content addressable memories, and a fixed length digest of a piece of data is used as index for retrieval of its attributes. Of course, due to the limited size of the digest and the potentially arbitrary data sizes, collisions are inevitable. Indeed, it is possible that different data share the same index, and a resolution strategy needs to be employed (e.g., linked list chaining or open addressing) [14]. Nevertheless, one benefit of this approach, relevant to this work, is that the use of hashing effectively obfuscates the relationship between the data attributes and their index tag. As a result, assuming that the underlying hash function is one-way, it is intractable to deduct the original data from their digests and identify their corresponding attributes.

III. THREAT MODEL

In order to put our obfuscation technique in the proper context, it is necessary to determine our assumed adversarial model. Our goal is to limit potential information leakage when branch decisions over encrypted data are resolved. Our main concern against this risk is an adversary with cryptanalysis capabilities that is able to extrapolate one or more bits of plaintext information by observing the outcome of runtime branch decisions, and at the same time have knowledge of the executed program.

In our model, the underlying concern is that even though a program processes encrypted data, branch decisions actually reflect their unencrypted counterparts. An adversary with cryptanalysis capabilities could test program execution over a large number of encrypted inputs and observe the distribution of branch outcomes. As a result, the adversary would be able to build a correlation between the branch criterion (for example, comparison with plaintext number zero), and the actual encrypted value. Furthermore, in cases when the branch criterion is also chosen (for example, comparison with a given plaintext value), this correlation could easily be expanded to all bits of the protected value through binary search.

In addition, branch decision outcomes may also be resolved through precomputed information. For example, branch decision tables can be used to statically determine branch outcomes for all possible ciphertexts corresponding to the applicable plaintext range (e.g., all 16-bit numbers). In such situations, adversaries that are able to offload these decision tables and use them as part of their cryptanalysis efforts, would also be able to decipher encrypted values, if these values are homomorphic. Indeed, cryptanalysis adversaries could apply homomorphic operations over a known range of values, and by leveraging branch table information they could perform frequency analysis and map the encrypted values to their plaintext counterparts.

To mitigate the threats discussed above, we assume the random oracle cryptographic model, where hash functions exist [15]. Furthermore, we assume adversaries with bounded resources and capabilities that are unable to reverse hash functions or build rainbow tables efficiently, as both those attacks are inefficient for today’s secure hash functions, such as SHA-3. In addition, allowing a trade-off between security and performance, our model assumes that adversaries are unable to efficiently reverse MISR-based signatures, which, as will be presented in Section V, allows further performance improvements compared to the more secure, hash-based, approach.

IV. OBfuscating Branch Decisions

As previously discussed, an inherent concern with branch decisions over encrypted data is that decision outcomes reflect the underlying unencrypted value. To mitigate this problem, we propose to obfuscate these decisions, by performing late branch resolution using the digest of a sequence of control values. Given that the applied digest is not efficiently reversible, branch decisions are no longer connected to encrypted values, but to a message digest over a sequence of values. Interestingly, this transformation is one-way and prevents leakage of any information about the values used.

In more details, our approach is formulated as follows: Let $BP$ be a branching program that executes a sequence of instructions $I_k$ for $k > 0$, and let $E_j$ for $j > 0$ be encrypted values processed by these program instructions. Since we assume $BP$ is a general-purpose program and not a simple function, there exists a set of instructions in $I_k$ that are branch instructions; we refer to this instruction set as $BI$, $n > 0$. Each instruction in $BI$ uses a piece of encrypted data from $E_j$ and determines the next instruction to be executed (i.e., the next $k$). Without loss of generality, we assume that each $BI$ makes the branch decision based on the sign of the plaintext
value corresponding to the encrypted piece of data in $E_j$. Each decision has two potential outcomes: $path-BT$ for branch taken, and $path-NT$ for branch not-taken.

Based on the formulation above, our approach performs obfuscation as follows: when a branch instruction from $BI_n$ is encountered, the program executes non-deterministically both outcomes in parallel, without resolving which branch path is correct. Specifically, the execution flow is divided in two paths (one is for $path-BT$ and the second is for $path-NT$), and both of them run independently until another branch instruction is encountered by both of them. Any changes made to the program state (e.g., memory value updates) are tied temporarily to each path, and are pending until the branch correctness is resolved at a later state (i.e., program state changes are committed only after the previously non-deterministic branch becomes deterministic).

Of course, the longer the non-deterministic execution continues, the larger the number of temporary parallel program states that need to be maintained. Without loss of generality and for simplicity of description, we assume that a branch decision is committed when a new branch is encountered by both alternative paths ($path-BT$ and $path-NT$). We call this 2-level non-determinism, as the control flow includes two branch nodes only. After the branch decision is resolved, the non-deterministic execution can continue from the deterministically committed path, and any program state changes would become permanent until that point. It is possible, however, to maintain more levels of non-determinism, at the expense of exponential growth in the program states required for late resolution of the correct execution path.

Our observation is that, by using $h$ levels of non-determinism, the actual execution path segment is defined by $2^h - 1$ control values respectively. In effect, branching is now linked to groups of control values (instead of single values), and this association can be obfuscated. In our approach, the obfuscated resolution of a branch decision requires an ordered hash digest of $2^h - 1$ encrypted values; each such digest is indexing a special hash table that stores the branch outcome as an attribute for all possible combinations of encrypted values. The number of elements (rows) in this hash table is linear to the number of $(2^h - 1)$ sized tuples over elements of set $E_j$, while the size of the branch outcome encoding is only $h$ bits.

Performance/Obfuscation tradeoff: One limitation of our approach is the need to execute multiple paths non-deterministically, which impacts the overall system throughput. In addition, maintaining hash tables with obfuscated branch decision encodings, as well as additional/parallel program states, increases the runtime memory requirements.

A. Aliasing Considerations

In our methodology, non-deterministic branch decisions are resolved and committed at a later execution step. Execution path resolution involves creating a message digest in order to index a hash table and retrieve an encoding of the correct path. This approach, however, would be problematic in case the generated message digest has aliasing (i.e., there are hash collisions). Indeed, a hash collision would cause uncertainty regarding the correct execution trail, as more than one paths in the control flow graph would match a given digest. Having more than one matches yields uncertain branch outputs, and thus it should be avoided. In practice, aliasing can be mitigated by using message digests of adequate length. As elaborated in our implementation discussion (Section V), we ensure that each message digest is unique.

B. MISR and SHA-3 Message Digests

In this work, we focus on two different methods to generate message digests of 3-tuples of encrypted values. The first is based on MISRs, and allows very fast and efficient hardware implementations, while the second is based on a general purpose hash function, and provides additional security guarantees in the random oracle model. The following paragraphs elaborate on how each digest is generated using our approach.

MISR-based digests: In this case, each encrypted value is sent to the input of the MISR sequentially. The latter needs to be configured with a sufficient number of cells (flip-flops) as to accommodate the input block; if the encrypted argument size is larger than the MISR size, it can be broken into blocks that are also processed sequentially. In this work, we selected an MISR of 64 cells that is based on primitive polynomial $x^{64} + x^4 + x^3 + x + 1$ [16]; this instantiation combines compact size with minimal aliasing probability. The initial state of the MISR can be set to zero (i.e., all cells are reset), and a different block is set at its input after one or more shift operations.\footnote{To minimize the probability of collisions, at least two shift operations are recommended for each MISR input block; for additional security, an optional secret key $sk$ can be used as the initial input in the sequence.}

Hash-based digests: For this option, we generate digests using the SHA-3 hash algorithm [17]. The hash input is an ordered concatenation of encrypted values, prefixed with an optional secret key $sk$ (for additional security), so that a single bitstring is processed. Furthermore, it is sufficient to use the 256-bit variant of SHA-3, as this size is adequate to ensure there is no aliasing in the generated digests. Even though SHA-3 digests are unique with overwhelming probability, and one-wayness is one of their security properties, generating such digests in hardware requires more rounds compared to MISRs.

V. ADDLEQ Case Study

We implemented our branch decision obfuscation approach in C, using a custom abstract machine for encrypted data, which is heavily based on the addleq single instruction language [4]. The selection of addleq as a case study for our implementation offers many benefits, namely Turing completeness, minimalism of description, branching as an integral part of every instruction, as well as a simple branch condition based on comparisons with zero. Nevertheless, our approach is also applicable to other abstract machines that support branching over encrypted values (e.g., [6], [18]).

A. ADDLEQ Abstract Machine

An addleq abstract machine executes each instruction as follows: first, it performs an arithmetic addition between two
memory locations, and stores the updated result back to the second memory location; then, if the updated result represents a positive integer, execution continues to the next instruction, otherwise it branches to a given target. To accommodate these steps, each instruction takes exactly three arguments (X, Y, and Z) that represent memory addresses, and the contents of address X are added to the contents of address Y. The branch decision (performed for every instruction), checks if the updated value in address Y is positive. The organization of an `addieq` program in memory is shown in Fig. 3.

This simple abstract machine can support encrypted values when additive homomorphic encryption is used for ciphertexts [19]. In our case, branch decisions performed during each step are obfuscated using 2 levels of non-determinism, which correspond to exactly 3 `addieq` instructions defined by their arguments \( \{X_1, Y_1, Z_1\}, \{X_2, Y_2, Z_2\} \) and \( \{X_3, Y_3, Z_3\} \). The first branch decision is normally based on the sign of the updated value at location \( Y_1 \), but this decision is deferred for after the second and third instruction are also executed non-deterministically. At that point, the values of memory locations \( Y_2 \) and \( Y_3 \) would also be updated (provisionally, as it is still uncertain which path is valid); finally, the program state is committed based on all three updated values at \( Y_1 \), \( Y_2 \) and \( Y_3 \). This non-deterministic execution is visualized in Fig. 4.

**Hash Table**: In order to properly decide which path to commit, a hash table needs to be precomputed when the program is compiled for the first time, and the input data are encrypted. A message digest for each combination of three encrypted values (i.e., 3-tuples) is generated to be used as hash-table index, and employing knowledge of the corresponding plaintexts at encryption time, the actual branch decision outcome is stored in the corresponding hash table entry. Specifically, for any 3-tuple of encrypted values \( \{E_1, E_2, E_3\} \) and optional key \( sk \), at the hash-table index equal to the message digest of the concatenation of these values \( \text{Digest}[sk||E_1||E_2||E_3] \), exactly 2 bits are stored representing the actual execution path inferred from the plaintexts corresponding to \( E_1, E_2 \) and \( E_3 \) (Fig. 5a).

The 2-bit encoding of the actual execution path can be determined using the plaintexts that correspond to the digested ciphertexts. Indeed, since branch decisions are essentially a sign test, an encrypted value corresponding to a positive plaintext selects the path of the non-taken branch (i.e., `path-NT`). To precompute the branch decision outcome over multiple levels, a binary tree structure needs to be envisioned: the root of the tree corresponds to the first branch decision, while every child node directly connected to the root corresponds to the next branch decision; each child has 2 child nodes as well.

Assuming that we apply obfuscation to 2 decision levels each time, we need to consider a subtree of 7 nodes (one root, two intermediate nodes, four leaf nodes). Then, our goal is to determine which leaf node represents a correct execution path after these 2 deferred branch decisions (one at root node, one at either intermediate node). Hence, for every combination of signs corresponding to possible plaintext values at these nodes (root and intermediate ones), the final execution path can be deterministically computed and used as attribute in our hash table. One important point, however, is that the hash table is indexed using hard-to-invert digests of encrypted values, while the stored attributes are just 2 bits of deterministic path encoding, computed from the sign of their plaintext counterparts. This effectively obfuscates any information about the corresponding encrypted values.

**Memory Deltas**: In our case study, non-deterministic execution can be achieved using the aforementioned binary tree data structure. Specifically, each node of the binary tree represents the memory deltas, which are the changes made to the program state by each branch path. Since both branch paths (`path-BT` and `path-NT`) are non-deterministically visited, the corresponding nodes in the binary tree would incorporate the memory changes (deltas) written back by `addieq` at memory locations \( Y_2 \) and \( Y_3 \) respectively. Since each `addieq` instruction updates exactly one memory location before making a branch decision, each memory delta stored in the binary tree nodes would contain exactly one memory value. As a result, the execution state can be reconstructed at the end of execution using the original program memory (which features a “dirty” bit to indicate a newer value in the binary tree), as well as the ordered set of memory deltas in the committed (deterministic) execution path (Fig. 5b).

**Tree Pruning**: At runtime, as soon as deferred branch decisions are committed, the binary tree storing the program state needs to be appropriately pruned. Any branch target that has been executed non-deterministically, but has not been committed, should be removed from the binary tree so that...
only one execution trail remains (except, perhaps, the last two non-deterministic branches). This binary tree state pruning needs to be executed after every other `addleq` instruction, since a new execution `head` is generated after the most recent path commit. In addition, since all branches in `h` levels are evaluated, a performance overhead of \((2^{h+1} - 2)/h\) times incurs at run time (i.e., \(3.0\times\) overhead for \(h = 2\) levels).

**B. Programming using Homomorphic Data**

For our implementation, we use homomorphic encryption from [20] to encrypt the data of `addleq` programs written directly in assembly language. Since only the data is encrypted, self-modifying code was not used in our experiments. Furthermore, our abstract machine was instrumented to perform modular multiplication instead of simple addition (i.e., a homomorphic operation), when the `addleq` instruction was referencing encrypted data instead of other instruction arguments. For the non-deterministic execution states, a binary tree data structure was used, offering convenient path commitments. Without loss of generality, and since programming `addleq` programs directly in assembly language is a tedious process, our experiments use contrived examples that perform selection of the maximum and minimum of a set of encrypted values. These values are provided as input data to the program, and the maximum and minimum values are stored at predefined output memory locations. We remark that there exist slightly more complex single instruction languages (e.g., `subleq`) that are compatible with our methodology and can benefit from cross-compilers to assist programming (e.g., [21]).

**C. Evaluation Results**

**Hash Table:** The computation of our hash table requires creating message digests for all possible 3-tuples of encrypted values. In this case study, we also assumed 8 bits for the unencrypted wordsize, and a 32-bit security parameter for encryption (i.e., encrypted values are 64 bits long). Thus, we have \(2^8 = 16.78\text{M}\) indices in our hash table, and depending on the digest method, we compute: (a) an MISR digest by processing the three encrypted values in each 3-tuple sequentially, and (b) a SHA-3-256 digest on the concatenation of those values (prefixed with key `sk`). Fig. 6 presents how the hash table scales with the unencrypted wordsize when 2 and 3 levels of non-deterministic branches are used. For comparison, if 3 levels were used, the number of entries becomes impractical, as illustrated in the graph. It should be stressed out, however, that computation of the hash tables (either using SHA-3 or MISR), is an embarrassingly parallel operation, and it is performed only once during initial encryption of data.

The attributes of each hash table index encode two bits that represent the actual branch to be committed after two non-deterministic branch decisions. Since `addleq` abstract machines make branch decisions based on the MSB of a control value (i.e., a branch is taken unless the corresponding plaintext is positive), this 2-bit value is precomputed once during initial encryption using the sign information of the corresponding plaintexts in each \((E_1, E_2, E_3)\) tuple. Moreover, it is sufficient to check 2-out-of-3 MSBs: if the sign of the plaintext corresponding to \(E_1\) is positive, only the sign corresponding to \(E_3\) is checked; otherwise, if the plaintext corresponding to \(E_1\) is negative, it is sufficient to check only the sign of the plaintext corresponding to \(E_2\).

**Obfuscation:** Our proposed obfuscation strategy is classified in terms of potency, resilience, and cost, using the standard evaluation criteria proposed in [22]. Potency corresponds to the increase in complexity and confusion introduced to human readers by the obfuscation, while resilience corresponds to the complexity of reversing the obfuscation and cost refers to the incurred obfuscation overhead. Potency is a metric based in part upon human cognitive abilities, and its qualitative scale ranges from low and medium to high and very high. Similarly, resilience ranges from weak, strong up to full and one way, while cost ranges from free and cheap to costly and dear [22].

Our MISR message digests are linear functions implemented through shift registers and exclusive-OR operations, so their cost is actually cheap (but not free). On the other hand, our SHA-3 digests are costly to implement, as 24 round operations are required, but still practical (both in hardware and software). In terms of resilience, SHA-3 by construction

---

**TABLE I**

<table>
<thead>
<tr>
<th>Metric</th>
<th>MISR</th>
<th>SHA-3-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potency</td>
<td>Very High</td>
<td>Very High</td>
</tr>
<tr>
<td>Resilience</td>
<td>Full</td>
<td>One-way</td>
</tr>
<tr>
<td>Cost</td>
<td>Cheap</td>
<td>Costly</td>
</tr>
</tbody>
</table>

Fig. 5. (a) Hash table used to retrieve 2-bit path outputs based on digests of arguments. (b) Binary tree structure used to resolve memory deltas.
is a one-way cryptographic function, while the multiple shifts with different inputs in MISR digests provide full protection, especially since the inputs are encrypted values (i.e., rainbow tables cannot be computed without access to the private keys). Furthermore, since both MISR and SHA-3 digests are human unreadable, their potency is very high. This classification is summarized in Table I for both digest strategies.

**Aliasing:** As also discussed in Section IV-A, message digests should have adequate bit size to ensure there are no collisions for all applicable inputs. In our experiments, we avoid collisions by using an MISR of 64 bits, where each encrypted value is applied to the inputs for exactly 2 shift operations. If a larger security parameter is used and encrypted values are longer than 64 bits, they are divided into 64-bit blocks, and each block is applied twice. In addition, since SHA-3 has a digest size of 256 bits, the probability of a collision is infinitesimal for the actual number of entries in our hash table.

### VI. RELATED WORK

The latest developments in fully homomorphic encryption and the invention of the Gentry scheme [23] have increased the academic interest in secure computation using encrypted values. In [24], the authors focus on universally composable security and demonstrate zero-knowledge and general function evaluation protocols, while [25] discusses verifiable delegation of computation on encrypted data using homomorphic hashing. Likewise, the authors of [26] demonstrate improvements in secure 2-party function evaluation, while in [27] a provably secure model for *retrievability proofs* is provided.

In the field of encrypted computation, the authors of [5] employ fully homomorphic encryption for secret program execution, while in [6] encrypted computation is performed using additive homomorphic encryption. Likewise, the authors of [18] leverage heuristic code obfuscation to enable a functionally complete set of operations on ciphertexts encrypted using the Paillier scheme [19]. Practical applications of encrypted computation include [28] and [29], where the security of the computation is reduced to the secure CPU container and a block cipher is used for performance.

### VII. CONCLUSIONS

The recent work in encrypted computation is indicative of the traction towards practical and efficient general-purpose computers that operate on encrypted values. In this work, we address the problem of information leakage due to branch decisions on encrypted data. Our approach is to obfuscate the direct connection of these decisions with the corresponding encrypted values, through MISR- and hash-based digests, used as hash table indices. Our *Addleq* case study evaluation indicates that this obfuscation approach has high potency and resilience, especially in the case of SHA-3 digests, by trading memory cost for maintaining hash tables and runtime cost for non-deterministic execution of all possible branches.

### ACKNOWLEDGMENT

This work was partially supported by the NYU Abu Dhabi Global Ph.D. Student Fellowship program.

### REFERENCES


