IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 16, 2016, accepted October 15, 2016, date of publication October 28, 2016,
date of current version November 28, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2622724

On the Security of a Universal Cryptocomputer: The Chosen Instruction Attack

STEFAN RASS AND PETER SCHARTNER
System Security Group with the Alpen-Adria-Universität Klagenfurt, Klagenfurt 9020, Austria
Corresponding author S. Rass (stefan.rass@aau.at)

ABSTRACT The ultimate goal of private function evaluation is the complete outsourcing of processing tasks to distrusted platforms (such as clouds), so that arbitrary functions can be evaluated without any leakage of secret information. Several successful concepts have been proposed in the past, the most striking one having been fully homomorphic encryption besides the well-known garbled circuits and multiparty computation. In this paper, we look at an idealized model of outsourced computation, which we call a *cryptocomputer*. This is a (theoretical) machine that works exactly like a real-life computer in the sense of understanding a standard assembly language, but retaining all its internal signals, registers, and memory encrypted at all times. The encryption is assumed under a key that is unknown to the attacker, and taken as secure (in any cryptographically meaningful way), so that no leakage of information from any ciphertext can be expected from programs with reasonable (polynomial) time complexity. Unfortunately, such a cryptocomputer is necessarily insecure, irrespectively of how the encryption looks like. In particular, we explicitly do not assume any specific form of security (chosen-ciphertext or other) or (a)symmetry of encryption; our attack works only on ciphertexts and makes no assumptions whatsoever on the encryption. We prove insecurity of the cryptocomputer by taking the encryption as a black box, and show how to decipher every signal in the computer by pure virtue of submitting proper instructions for execution. Our attack falls into the general category of side-channel attacks, however unlike other related attacks, does neither exploit physical nor any logical characteristics of the underlying platform (besides the execution flow being observable). Somewhat surprisingly, it turns out that although the problem that we consider is cryptographic, it seemingly has no cryptographic solution and apparently calls for an interdisciplinary approach from new directions.

INDEX TERMS Private function evaluation, security, applied cryptography, side-channel attack.

I. INTRODUCTION

Assume a standard computer (laptop, smartphone, etc.), following a conventional von Neumann or Harvard architecture, and understanding a reduced or complex instruction set (RISC or CISC). Furthermore, suppose that some encryption or other disguise is applied to the internal signals so that nothing about the inner data can be learnt from knowledge of the signal and the public parameters (of the protection). For simplicity, let us think of the internals being protected by some encryption scheme (the case of threshold cryptography and multiparty computation (MPC) is discussed in the concluding section of this work).

It is a well known fact that any reasonably powerful model of computation requires at least the functionality of a Turing machine, or equivalently, a quite small set of instructions that cover the following:

- Arithmetic and logic instructions: an increment instruction (like `C++` in the language of the same name) is already sufficient here, but practical assembler languages offer at least additions, subtractions, multiplications, divisions and bitwise logical operations.
- Conditional branching: at least a while-statement is necessary, alternatively also an if-statement. Omitting both leads to loop programs, or primitive recursive functions, the latter of which are known to be unable to compute the Ackermann function).
- Memory access: although this does not add anything to the expressiveness of the computational model, it is nevertheless a practical necessity, especially in light of big data applications and the need to process possibly huge amounts of data. Many assembly languages offer various kinds of addressing modes, such as direct, indirect (pointer arithmetic), etc.

For illustrative purposes (only), we will assume an assembly instruction to work on registers of fixed size (32 bit or 64 bit are common choices), and to act on encrypted content

at all times. For the syntax, we will stick with the Mips assembly language [1], in which registers can be marked by a preceding $-symbol. Under this notation, an instruction like `add $s0, $s1, $s2` would compute the sum of the values in `$s1` and `$s2` and put it into register `$s0`. To make our assumed computational platform more precise, Figure 1 illustrates a simplified version of a datapath that is – in variations and with refinements – found in many practical microprocessors. The central point of our consideration is the processor working by a repeated sequence of instruction fetching, instruction and operand decoding, execution, and writing results back to the memory. The data path illustrated in Figure 1 therefore shows only the main components involved in these steps, which in particular includes an instruction memory (homed together with the data memory in a von Neumann architecture, or physically separate from the data memory in a Harvard architecture), a register file, whose content is fed into the arithmetic-logic unit (ALU), and afterwards written back to the memory. We explicitly added the increment of the program counter by 4 bits (in a 32 bit processor architecture), with an optional direct feedback of data into the program counter (PC), upon a conditional jump (triggered by the ALU in the execution phase). The multiplexer (MUX) shown in the picture either updates the PC with the direct increment (sequential flow of instructions) or with the jump target (if the ALU triggers this by the signal labeled as "conditional jump?" in Figure 1).

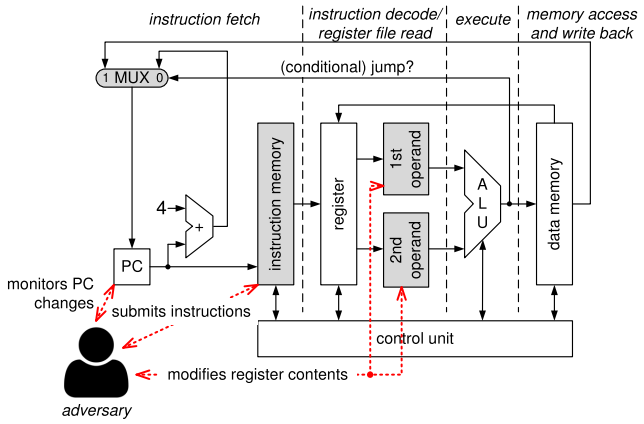

FIGURE 1. Standard Data Path (simplified) – Computing Platform.

We believe that this simplified diagram resembles the skeleton that underlies most practical computing devices today, and especially suits the execution of assembly code. The grayish boxes shown in the picture are parts that the adversary can observe or control, assuming access to the data path. Practically, it is often not even necessary to fiddle with the processor internally, since the code can be supplied externally, and written such that registers get filled with the desired content. We will postpone the adversary model until later, and first describe the cryptographic protection that we assume being applied in the data path; hereafter referred to as *computing platform*.

Throughout this work, we will (implicitly) assume that the actual value within a register, say `$s1`, is inaccessible to the computing platform due to an encryption, but that the platform is capable of executing the assembly instruction. Fully homomorphic encryption (FHE), as one example of how this can be done, directly offers an addition and multiplication operation to work on ciphertexts, while the actual operation is hiddenly applied to the inner plaintext. Alternatives like garbled circuits (GC) achieve the same functionality, based on a circuit representation of the instruction, and MPC produces the result interactively between many parties, while leaving the actual result up to a reconstruction by a designated party. In the following, we will not adopt any specific assumption on *how* an instruction is executed, except for an illustrative sketch given in section III and the assumption that the operands of an instruction are securely enciphered. However, the encryption scheme will remain a black box at all times, and can be assumed to be secure under any notion (indistinguishability under active attacks, or similar).

We will adopt the following semi-formal understanding of a universal cryptocomputer (based on the data path sketched in Figure 1) to reflect the "ideal" setting of private function evaluation:

Definition 1: A *universal cryptocomputer* is a computing machine that:

1) understands an assembly instruction set (RISC or CISC)
2) can execute arbitrary programs (universality), which are stored in plain form (except for data within the code that is encrypted, such as constants),
3) works on ciphertexts only, and never decrypts any of its internal signals.

Definition 1 implicitly assumes that the program counter remains available in plain, as is required to retrieve the next instruction after having completed one. The central insight in this work is that a universal cryptocomputer meeting the described requirements (of definition 1) is *insecure*, since:

- No matter how an instruction is executed or how the encryption looks like, the capability of the execution platform to execute *any instruction of our own choice* can be exploited to disclose the hidden register contents (and therefore decipher also the memory content). We call this the *chosen instruction attack*, and describe its details in section IV.
- Enlarging register or key sizes provides no improved security, since a ciphertext's size is necessarily polynomial in the security parameter. On a sufficiently powerful assembly language (offering additions or shifts), we obtain a linear time algorithm (linear in the security parameter) to disclose a ciphertext bitwise. Details are given in section IV-B.
- Since our attack presumes that arbitrary code can be written and submitted for execution, we could suspect code obfuscation to help us out. Unfortunately, section IV-C shows how assembly mnemonics can be deobfuscated (using a probabilistic method).

Although our presentation will be given in Mips assembly syntax, the risks are not mitigated by switching to other assembly languages, since the common subset of instructions that all languages need to offer (only in different forms), can be used to launch the attack only in adapted form (section V). Section VII elaborates on possible countermeasures and opens avenues for future work. Conclusions are drawn in section VIII.

II. RELATED WORK

Private function evaluation is a special case of the more general concept of functional encryption [2], [3], which in particular includes specialized constructions like attribute and predicate based encryption [4], [5], searchable encryption [6], [7], and similar. The problem of transferring the Turing machine model into an entirely encrypted context has been treated in [8]–[10], and Mips assembly code has explicitly been investigated in [11]. Practical computational platforms that allow for arbitrary computations are available in various forms like general purpose platforms (ObliVM [12], or HEROIC [13]), or designated compilers that transfer a function's description into an executable representation (usually a circuit, evaluated either as a garbled circuit [14] or using somewhat homomorphic encryption [15], [16]). A similar notion is presented in [17], where a client can send an obfuscated program to some distrusted server, who evaluates it and learns nothing except the program's output. The last reference is particularly interesting in our context, since it may (although not yet verified so) provide hints on how the described attacks in this work could be thwarted. Closely related to our work is also the one of [18], who analyze security on the programming language level (concretely talking about `C`), to compile a garbled circuit. Although many security notions have been tailored to the specific challenge of private function evaluation (e.g., *circuit privacy*, which asks for the function f itself to remain hidden from the evaluator's eyes), the exploitation of the "universality" of the processing seems to be an unreported possibility so far. Consequently, attacks like the one we describe in this work are not covered by the usual notions of security in the context of private function evaluation. One notable exception to this is MPC [19], which admits the execution of arbitrary instructions as an interactive process between several parties, while keeping the inner information shared and thus unrecoverable by any party. Secrecy is bought here at a considerable communication (network) overhead. Also, MPC does at some stage require a "combiner" (possibly composed from several cooperating instances), who recovers the results ultimately. This is the point where our attack becomes mountable again.

III. PRELIMINARIES

A general asymmetric encryption scheme is a triple (KG, E, D) of three algorithms with the following meaning:

- **KG**: a probabilistic algorithm taking a security parameter $t \in \mathbb{N}$ to output system parameter and a public and private key pair (pk, sk).
- **E**: an algorithm that takes a message m and public key pk to output a ciphertext c.
- **D**: an algorithm that takes a ciphertext c and secret decryption key sk to recover a message m'.

Without loss of generality, we will formulate the upcoming considerations in terms of asymmetric encryption, leaving the use of a symmetric scheme as an obvious change of no effect.

Furthermore, let us assume that the register size is being compatible with the block size of the encryption (alternatively, assume the encryption to be a stream cipher). This assumption is made only for technical soundness, but has also no explicit impact on the upcoming considerations.

To get an idea of how assembly instructions can be executed on encrypted ciphertexts, let us – for the moment – take the register size to be quite small, say only one bit. Then, it is trivial to execute an addition instruction like `add $s0, $s1, $s2` by using a humble lookup table of size 2×2 (since each register can take on only two values). Larger register sizes (say 32 bit) would consequently cause an exponential blow-up of the lookup table (having size $2^{32} \times 2^{32}$ for example), which makes this approach clearly impractical. However, note that many schemes of private function evaluation (except multiparty computation) indeed work on small blocks (even single bits, e.g., garbled circuits).

The task of executing an assembly instruction by a table lookup is (assuming the table to be of practical size) a matter of matching the operands `$s0`, `$s1` to the row and column of the lookup table, from which the encrypted result can be retrieved. Figure 2 illustrates this idea for a simple addition instruction. The lookup table is indexed on the two operands of the instruction, both being ciphertexts (here denoted as u_i and v_j). Note that the decryption as indicated in the figure does actually *not* happen in reality, but is denoted only for clarification.

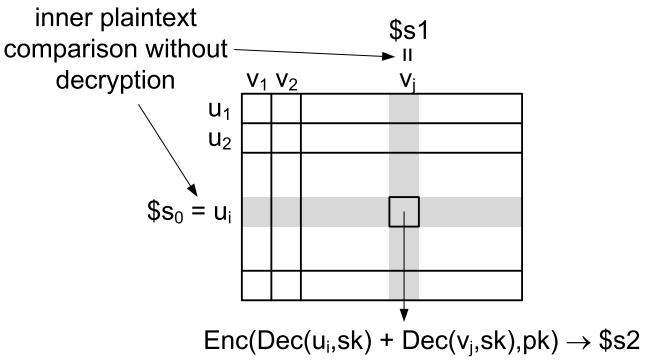

FIGURE 2. Execution of `add $s2, $s0, $s1` using lookup tables.

Although security being defined as indistinguishability explicitly prevents such a comparison, a specific construction of a *public key encryption with equality test* (PKEET) has been given [20], and even generic black box constructions that endow any given asymmetric encryption scheme (KG, E, D) with a ciphertext comparison facility are also available [9]. So, w.l.o.g., we may assume such an extended encryption scheme to be used for the assembly interpreter,

which then extends the three algorithms above by two more:

Aut: an algorithm that upon input of the system parameters and keys, outputs a *equality testing key* ek. This key is explicitly useless for decrypting ciphertexts, but lets us check equality of inner plaintexts.

Com: an algorithm that takes two ciphertexts c_1, c_2 and the equality testing key ek to and outputs

$$\mathsf{Com}(c_1, c_2, ek) = \texttt{true} \iff \mathsf{D}(c_1, sk) = \mathsf{D}(c_2, sk).$$

Several remarks about this extension are in order here:

1) Both, the authorization and comparison algorithm can be designed to work on ciphertexts encrypted under different public keys. Then, ek is a joint key from two sources, i.e., expressing their joint consent to compare ciphertexts (controllable authorization).
2) The key ek in the above instance of assembly code execution on encrypted data acts as an *evaluation key*. Such keys are also found in instances of FHE [21].
3) Most importantly, we will use the comparison oracle Com only for describing the chosen instruction attack in an initial form. In section IV-A, we abandon this functionality, and return to the original standard encryption scheme (KG, E, D) that disallows comparisons. Thus, only to make a start, we will endow the computing platform with a flaw in the form of the comparison oracle, but later on show how such an oracle can be constructed in a completely clean and well-designed platform that uses a conventional (unmodified) encryption.

a: ADVERSARY AND SECURITY MODEL

Our adversary is assumed to have full access to the execution platform (indeed, the adversary is who runs the code), but without access to any of the keys that have been used to encrypt the data. Furthermore, we assume the attacker to be computationally bounded according to the requirements of the underlying encryption. This assumption is generic and depends on the concrete choice of (KG, E, D), but in most cases will amount to restricting matters to polynomial time-complexity (in the size of the keys, resp., the security parameter that went into KG). More formally, we define the adversary to be/run a probabilistic algorithm A, with

1) *inputs* being every signal and register content available in the data path (computing platform), but this access being on ciphertexts only, and
2) *oracle access* to the instruction memory, register file (specifically the operand register contents), and the program counter value (these accessible parts are highlighted in gray in Figure 1), and
3) *polynomial running time* in the variables t, N and M, where:
   - t is the security parameter that went into the KG algorithm to initialize the encryption.
   - N is the number of instructions in the assembly language
   - M is the number of registers

   Note that we take the running time as independent of the size of the data memory (the latter can safely be assumed as polynomially sized in t, since otherwise, a super-polynomial lot of ciphertext would be available, in which case most security notions for encryption become void).

Security is here (qualitatively) understood as the attacker's inability to learn anything about the underlying data (with efforts taking polynomial time in t, N and M). As we will show, it is not necessary to become any more specific or formal on the security definition here, as the attacks described will entirely disclose the secret data into its plain (unencrypted) form (quite efficiently), under the assumptions stated above.

IV. THE CHOSEN INSTRUCTION ATTACK

In its simplest form, consider a `for`-loop of the (generic) form `for i = 1 to n do`...*something useful*...`endfor`. In this bit of code, n is the variable of interest. Then, simply observing how often the loop is executed directly delivers the value of n, no matter of whether n is encrypted or not. An obvious countermeasure is having the compiler add a random number of dummy iterations to hide the actual value of n. However, achieving this for every program in a meaningful way is intricate and increases the complexity of the compilation process.

The actual problem, however, is not resolved by any such precautions, if the assembly instruction interpreter allows for arbitrary instructions to be executed.

To see this, let us return to Mips assembly as our illustrative example (stressing that other assembly languages offer the same functionality only in different syntax). For the example, let `$c` be the register whose content is encrypted and shall be disclosed. Furthermore, in the simplest case, assume a comparison algorithm Com to be available. A chosen-instruction attack can be mounted along the following steps:

1) Subtract the value `$c` from itself, giving an encrypted version of zero, irrespectively of the value hidden in `$c`. In MIPS assembly language, execute the command

   ```
   sub $s0, $c, $c,
   ```

   so that `$s0` $\leftarrow \mathsf{E}(\texttt{\$c} - \texttt{\$c}, pk) = \mathsf{E}(0, pk)$.
2) Next, invoke $\mathsf{Com}(\texttt{\$c}, \texttt{\$s0}, \mathsf{ek})$. If this returns `true`, then `$c` $= 0$ has been discovered and we are done.
3) Otherwise, `$c` $\neq 0$ means that we can divide `$c` by itself to get an encryption of 1 for the quotient and another encryption of zero for the remainder (a byproduct that is not required at this point). In MIPS assembly, we would thus submit the instructions

   ```
   div   $c, $c
   mflo  $s1   # $s1 ← E($c/$c, pk) = E(1, pk)
   ```
4) Now, given `$s0` $= \mathsf{E}(0, pk)$ and `$s1` $= \mathsf{E}(1, pk)$, it is a trivial matter of executing

   ```
   add $s0, $s0, $s1
   ```

for k times to get $\texttt{\$s0} = \mathsf{E}(k, pk)$ for $k = 0, 1, 2, \ldots$.

5) Repeating the previous step, and by virtue of Com, we can brute-force disclose any unknown register content, given that the registers are only 32 or 64 bits, leaving us with no more than 2^{32} or 2^{64} cases to test. This is feasible on nowadays computing architectures.

It must be stressed that under a standard implementation of public key cryptography, the adversary would be given the public key pk, which quite directly would allow to produce the sought values for the comparison. This is indeed a valid shortcut to the above attack, and also applies to any deterministic asymmetric scheme like RSA over small plaintext spaces (simply trial encrypt all candidate plaintexts and see if the given RSA ciphertext is reproduced). In the following, let us therefore strengthen our hypothesis on the adversary to *be unable* of producing ciphertexts on its own (say, if the encryption key pk is never made public). This indeed generalizes the upcoming considerations, as this assumption includes all symmetric encryption schemes into our description.

Since the attack in the described form assumed the existence of a comparison algorithm Com, let us now modify it to work without this assumption.

A. GETTING RID OF THE COMPARISON ORACLE

The previously described attack did assume the existence of a comparison oracle that shipped with the encryption function $\mathsf{E}(m, pk)$, but assumed nothing on the structure or security of the encryption itself. Let us now drop this extension and confine ourself to a black-box usage of the encryption scheme (KG, E, D) in its plain form. Indeed, it can be shown that the comparison function can be established even without changing the encryption scheme at all. So, we will modify the above attack to work without any assumptions on the underlying encryption.

Various options are available to construct a comparison oracle, such as (at least):

- Use branching instructions: for example, Mips assembly offers a branch-if-equal command `beq $s0, $s1, label`, which upon execution can be observed to reset the program counter in case of equality. This directly implements the sought comparison oracle. Alternative comparisons like $\neq$, $<$ or $>$ can in combination be used to yield the same output (for example, $a = b \iff \neg(a < b) \wedge \neg(a > b) \iff a \leq b \wedge b \leq a$, and $a > b \iff b < a$, so that only one conditional branching instruction would be required).
- Use memory access patterns: loading a memory cell into a register is a matter of issuing `lw $s1, offset($s2)`, which accesses the memory at the cell with address `$s2` + `offset`. Enforcing `$s0` to contain zero (as described previously), and using this ciphertext for `offset`, we end up accessing the address $0 + \texttt{\$s2} = \texttt{\$s2}$. The equality check on the encrypted registers `$s2` and `$s3` are then plainly doable by checking if the instructions

```
lw $s1, offset($s2)
```

```
lw $s1, offset($s3)
```

with `offset` = E`(0,pk)` access the same memory cell.

- Spare the comparison at all by exploiting error flags and status registers. For example, if a division by zero causes a change in some status register, then this change could be observed as an indication of a zero division. Likewise, doubling a register until an overflow is indicated works equally well. A countermeasure against this is, however, easy if the status registers are re-randomized whenever an assembly instruction is executed.

B. LARGER REGISTERS OR KEYS DO NOT HELP

As we did not assume anything on the encryption, this also means no assumptions on the key sizes. So enlarging the key space cannot help here. Unfortunately, working with larger registers to prevent the final brute-force opening is not effective either, since the attack can still be mounted.

To see this, let us jump into the attack at the point where an encrypted value zero and one have been constructed, and assume that the assembly language offers the logical bitwise `and` operation. As before, let $\texttt{\$s1} = \mathsf{E}(1, pk)$ be available, and let `$c` be the register content to be disclosed. Then, issuing the command

```
add $s1, $s1, $s1
```

for $k = 0, 1, 2, \ldots$ times will fill the register `$s1` with the value sequence $\mathsf{E}(2^k, pk)$. Then, executing

```
and $s2, $c, $s1
```

and testing the result `$s2` to equal zero is a straightforward mean of testing the individual bits in `$c` to be zero or one. This process requires only k repetitions of the procedure to entirely disclose a k-bit register, and thus remains efficient even if 128 Bit, 256 Bit or larger registers are being used (that are not accessible through brute-force trials like described above).

Alternatively, the idea of doubling a register content can be used to left-shift the content until an overflow is flagged. Each time when the content is shifted to the left by 1 bit (by doubling the content via `add $c, $c, $c`) an overflow indicates a 1 at the most significant position, while no overflow tells us that the most significant bit has been a zero.

The same possibility is opened by logical shift instructions. Especially the right-shift can be used to produce a zero-value, if it is issued at least k times on the same register, when k is the bitsize of the CPU.

C. REVERSING CODE OBFUSCATION

Since our attack heavily relied on using the assembly code in a known form, obfuscating the op-codes appears as a straightforward mean to make the attacks at least more difficult (note that we cannot obfuscate the operands easily, since the interpreter has to know where to retrieve the data from). This is, however, only of limited use, since many instructions

can be recognized upon their syntax (for example, arithmetic and logic instructions take three registers as parameters, as opposed to conditional branches taking two registers and a label). In the following, we consider code obfuscation in simple terms of randomly renaming mnemonics while leaving the semantic unchanged. Full code obfuscation that induces dummy instructions and decoy code structures provides no additional security against our attacks, since the goal is here not on reverse engineering the code, but merely on identifying the instructions required to launch a CIA attack.

Unfortunately, it is exactly the aforementioned issue that helps discovering which instruction does what. Before describing this in section IV-C.1, let us first describe how the required arithmetic instructions (subtraction, addition, division) can be discovered or emulated.

In the following, the attacker may take any pair of distinct registers or memory cells at any time of a normal program execution; let us call them `$s0` and `$s1`. With high probability ($\approx 1-2^{-k+1}$ for a k-bit register), the two registers will have different and nonzero values.

1) DEOBFUSCATING BRANCHING MNEMONICS

Let the obfuscated version of a conditional branch be denoted as `cb $s0, $s1, label`, where there is no information on whether `cb` tests for $=, \neq, <, >, \leq$ or $\geq$. Furthermore, let `$s0` and `$s1` be two arbitrarily picked registers whose contents are (most likely) distinct (to assure this, the following steps should therefore be repeated with several choices of `$s0` and `$s1`).

Figure 3 illustrates the steps to disclose (or also program) a comparison for $\leq, \geq$ or $=$, using an unknown conditional jump instruction `cb`. The flowchart-like diamond boxes have a Y-branch if the instruction in the box caused a jump, and go into the N-path if the conditional branch `cb` did nothing. The intermediate rectangular boxes list the possibilities left at each point in the flow.

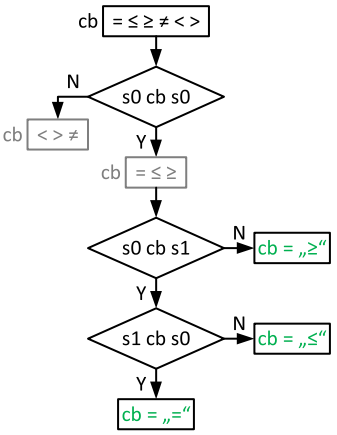

FIGURE 3. Revealing the semantic of `cb` $\in \{\leq, \geq, =\}$.

Once an instruction `cb` is found that branches conditional on $=, \leq$ or $\geq$, the control flow in Figure 3 can be used as a template to implement the necessary comparisons for the chosen instruction attack.

This in turn lets us create the values 0 and 1, which can then be used to discover tests for strict inequality ($<, >$ or $\neq$). Again, let `cb` be any of these tests (as indicated by the top left N-branch in Figure 3). Also, let `$s0` and `$s1` now be two registers that contain the (encrypted) values 0 and 1. Then, consider the execution of the following two instructions:

```
cb $s0, $s1, label1   # 0 < 1?    (1)
cb $s1, $s0, label2   # 1 < 0?    (2)
```

We can distinguish three cases:

1) Both instructions cause a jump: then `cb` tests for $\neq$.
2) Only one of the instruction branches: then `cb` is either for $<$ if (1) jumped to `label1`, or for $>$, if (2) jumped to `label2`.
3) The third case of neither instruction jumping is impossible, conditional on `cb` being either $\neq, <$ or $>$. Otherwise, it indicates a mistake in the first discovery along the flow in Figure 3. In that case, our choice of `$s0` and `$s1` was unlucky, and we must start afresh.

2) DEOBFUSCATING ARITHMETIC/LOGIC MNEMONICS

Let the unknown instruction be written as `cmd`. In case of Mips assembly, the instructions for division and multiplication are recognizable by taking only two arguments, so we exclude them here in this particular illustration.

Let us assume, w.l.o.g., that the instruction set contains a logical shift to the right, then shifting a register k times to the right surely creates a zero value (a left shift works equally well, and can even be emulated by repeated additions to double a value; see above).

If, in addition, a subtraction instruction is available (or an addition of the twos-complement), then we can execute both instructions and then do a conditional branch to check if they give the same result. To disclose the subtraction instruction, we write code of the following (general) pattern:

- Let `cmd1`, `cmd2` be any two instructions, which we suspect to be a shift and a subtraction (not knowing if this is correct or which is which). Then, use `cmd1` and `cmd2` to produce a value suspected to be zero on the two ways as described above.
- Use the conditional branching for equality testing to check if the two previously obtained results are the same. If so, then we know that {`cmd1`, `cmd2`} = {`sub`, `shr`}, albeit we still do not know which is which.
- The final distinction can be made by re-doing the comparison after only a single invocation of `cmd1` and `cmd2`. The result will be zero only for one of the two (assuming that the ciphertext on which we were working was not zero or one by coincidence), indicating the one that produced the zero to be the subtraction instruction, and leaving the other to be the right shift.

The above procedure needs to be repeated, obviously, for every pair of instructions, although step IV-C.2 is only to be done once. The overall effort is in $O\left(\binom{N}{2}\right) \subseteq O(N^2)$ for a total of N instructions. This is indeed feasible, since

even in a CISC architecture with an assumed number of, say 400, instructions, we end up testing a total of no more than $\binom{400}{2} = 79800$ cases. On a RISC architecture (like Mips) with 30 instructions, there would be only 435 pairs to be tested.

Also, note that the procedure has a general chance to fail, in case of unfortunate choices of the contents put into the registers `$s1` and `$s2` (remember that we allowed the attacker to act probabilistically). Thus, the overall deobfuscation should be repeated to lower the error sufficiently. Indeed, for M registers available, there are $O(M^2)$ pairs that can be used for the attack, which bounds the number of trials possible for the adversary. Additionally, the attacker could apply any arithmetic instruction, albeit unknown in its functionality, to modify a given ciphertext into something else (and most likely different to the previous value).

Having discovered the subtraction instruction and with an equality test being doable by the previously discovered conditional branches (upon $=, \leq$ or $\geq$), then it is easy to issue a subtraction between two (arbitrarily chosen) register contents, followed by an addition and a check if the suspected addition reversed the effect of the subtraction. This incident reveals the mnemonic behind which the addition instruction is hidden:

```
sub  $s0, $s1, $s2  #$s0 ← $s1 − $s2
cmd1 $s3, $s0, $s2  #$s3 ← $s0 + $s2
if $s0 = $s3 then cmd1 = add
```

Finally, even if the assembly language does not offer a direct command for divisions, it is an easy matter to emulate a division by repeated subtraction. Alternatively, if the previous investigation has dug up a right shift instruction, then repeatedly shifting a content to the right and testing for the value to become zero leaves us with the butlast intermediate result to be necessarily equal to 1. Hence, this can replace the division in the original CIA-attack.

Recognizing other operations may be equally possible using known relations that these satisfy, such as $\$s0 \times \$s0 = \$s0$ if and only if $\$s0 \in \{0, 1\}$. Likewise, the only logical self-inverse two-operand instruction is the XOR, while the logical not is also self-inverse (an involution) but takes only one operand. For testing the zero, even without a subtraction, we can exploit the fact that $\$s0 = 0 \iff \$s0 + \$s0 = \$s0$, etc.

V. USING OTHER ASSEMBLY LANGUAGES

Not surprisingly, many CPU architectures provide a sufficient lot of instructions that allow mounting the described attacks. We give only three examples out of the vast amount of architectures:

- Intel 8051 CISC architecture [22]: this provides the `djnz`-instruction, meaning *decrement-and-jump-if-not-zero*. This is a quite convenient and even direct way of brute-force testing a register content; simply write a loop to decrease the inner (encrypted) value until the instruction no longer jumps. Counting the iterations then directly delivers the inner value without any decryption.
- x86 processor [23]: the original 8086/8088 instruction set already has all in it that we need, since there is a command `SUB` doing subtractions, and an `INC` instruction that directly gives us an (encrypted) 1. Conditional jumps are offering all sorts of comparison.
- ARM processor family [24]: these also have a `BEQ` instruction that can be used directly, besides offering the necessary arithmetic through the instructions `SUB` and logical shift to the right (`LSR`).

Summarizing our findings in a comparative way, Table 1 shows the instructions that each platform provides (at least) to enable a chosen instruction attack. Any platform providing similar instructions allows for the same (or similar) CIA attack. The table is not exhaustive and shows only selected instructions towards illustrating the diversity of instructions and highlight common instructions that enable a CIA attack.

TABLE 1. Examples of instruction set vulnerability to the CIA attack.

architecture	CIA-enabling instructions (selection)
MIPS	add, beq, and, sll
x86 (8086/8088)	SUB, INC
Intel 8051	djnz
ARM	beq, sub, lsr

VI. ATTACKING CIRCUIT-BASED SCHEMES

Many schemes in the literature (GC, FHE and also MPC), prescribe a conversion of the program (algorithm, function) f into a proper representation for evaluation. For GC and FHE, this representation is a circuit. The conversion requires a "compiler" that takes a description of f to output the proper circuit representation. This conversion, to be practically feasible, requires the circuit to be uniform, i.e., constructible by a Turing machine, or more practically, by an algorithm that takes an integer n and a description of the function f to construct a circuit that computes f on all inputs of (exact) length n (see [25] for the theoretical background). Practically, this conversion is demanded in polynomial time (to be feasible), which fits into the computational bounds that we imposed on our adversary in section III.

A circuit family, however, also entails that for data lots of varying length, circuits have to be compiled afresh, which somewhat deviates from our initial requirement of the computing platform to be universal (in the sense of being capable of running *any* code on *arbitrary* inputs). Nevertheless, circuit models of private function evaluation can be used to realize core parts of the computing platform like the arithmetic-logic unit (ALU), or to represent the data path in its entirety as one large circuit (as is the case for real-life microprocessors).

Even despite the (compiled) "circuit-code" for f being hardly changeable towards a different functionality, chosen instructions can nonetheless be executed upon making the circuit compiler (such as proposed by [26]) into an integral part of the execution platform. That is, it is conceptually no difference to declare the last two parts of the compilation

chain

$$\text{Program} \to \underbrace{\text{Compiler} \to \text{Evaluation}}_{=\text{ execution unit (ALU)}}$$

as the new platform, which then again takes an arbitrary program (instructions) to execution (only converting instructions into the proper circuit to evaluate them; but our attack does not hinge on ''how'' an instruction is executed, whether using a circuit or otherwise). If the program counter remains unencrypted, say to enable instruction fetching, then the chosen instruction attack can be launched as described before. Otherwise, if the circuit is constructed to utilize all signals (irrespectively of the inner computational execution flow), then a CIA attack can be mounted if the attacker gains oracle access to the decryption function, by virtue of which it can run code whose execution flow becomes visible in the decrypted output. Again, such an implementation of private function evaluation is outside the definition of the universal cryptocomputer that we described above, but shows that notions like chosen ciphertext security (which usually assume oracle access to a decryption algorithm), do not generally cover chosen instruction attacks too.

If the computational platform uses multiparty computation, the story is more involved. For example, the representation would be a protocol between separate parties, all of which would need to cooperate on executing the adversary's chosen instruction(s). This requires the adversary's control over all computing players, which is usually precluded in the context of MPC. Extending our attack to cover MPC implementations thus remains an interesting aisle of future research.

VII. COUNTERMEASURES

There appears to be neither an obvious nor simple countermeasure against the attacks, due to their simplicity and generality. As was argued in section IV-C, obfuscating the code could be reverted. One possibility (yet unexplored and as such a matter of interesting future work) is making execution traces indistinguishable [17]. Roughly speaking, we may think of this as applying a transformation to the original code, in order to make a program use each execution path equally likely. However, such a precaution would either be applied a priori (before submitting the code for execution, cf. [27]), or a posteriori (i.e., after the code is received by the platform for execution). In any case, the attacker was assumed to run the execution platform with unlimited access to its internals, so we cannot safely rely on such a countermeasure to be in place when the platform is distrusted.

Picking up the idea of a making a program ''non-malleable'' we could attempt to link instructions to one another so that one instruction cannot be executed without the preceding statement having executed immediately before. Unfortunately, the issue of achieving this ranges much beyond a simple chaining by hashing or digital signatures, since the dependency must be anchored in the data that is executed (which in turn is inaccessible to the cryptocomputer). Although ideas like these already exist in an (admittedly humorous context of esoteric programming languages like Malbolge [28]), a concrete technique to make instructions data-dependent on one another appears unavailable, and opens another interesting avenue of future research. Some progress into this direction, however, has been made by [29], which considers instruction overlapping. Whether or not this technique can be extended to suit our needs is an open issue. Besides, measures used to provide control flow integrity (CFI) [30]–[32] may also be useful and will be included in future research.

Finally, we can relax the assumption on the plain program counter to be available. Suppose that every instruction is such that it, as an auxiliary piece of data, carries information on where to fetch the next instruction (e.g., every instruction carries a jump). Although the code can then be presented ''out of order'', it would be easy to put it back into its original sequence, unless the next instruction pointer is encrypted. This induces the possibility of using private information retrieval [33] or oblivious RAM to fetch the next instruction. This could – theoretically – disable the adversary's chances of observing the execution flow ''in plain'', but come at the cost of having to sweep over the entire memory (including data and code), in order to retrieve one single instruction (note that we cannot resort to treating the program only, since plain memory access, even if it retrieves ciphertexts only, would then let us recover the functionality of a comparison oracle; cf. section IV-A). The practical overhead induced by such a measure, however, is expectedly large, even though quite simple suitable schemes exist [33], [34]

VIII. CONCLUSION

This work is intended as a step towards discovering some theoretical limitations of cryptographic techniques in terms of applicability to private function evaluation. When encryption shall be applied to a contemporarily common computing architecture without adapting the architecture itself, then our results show that pure encryption is intrinsically incapable of solving the problem to the final extent, so that physical protection, hardware security and non-technical measures remain inevitable. Within a standard data path structure such as underlies today's microprocessors, there seems to be no obvious way to escape the problem, because retracting necessary instructions from the assembly languages either makes the resulting language impractical (or inefficient), or does not remove the CIA attack issue. Also, side-channel information leakage appears to be somewhat unavoidable, since it either occurs logically (through the CIA attack), or physically, induced by the specific hardware implementation. Circuits as a computational model appear cumbersome in this regard, since the circuit has to be crafted specifically for a fixed function. Mitigating the problem therefore seems to call for completely different and new computing architectures.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for valuable suggestions that have helped to improve the paper, and also pointed to interesting future research questions.

REFERENCES

[1] Imagination. (2015). *MIPS32 Architecture*, accessed on Mar. 24, 2015. [Online]. Available: http://www.imgtec.com/mips/architectures/mips32.asp

[2] D. Boneh, A. Sahai, and B. Waters, "Functional encryption: Definitions and challenges," in *Proc. 8th Theory Cryptogr. Conf. (TCC)*, Springer LNCS, vol. 6597. 2011, pp. 253–273. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987260.1987281

[3] B. Waters, "Functional encryption: Origins and recent developments," in *Proc. 16th Int. Conf. PKC*, vol. 7778. 2013, pp. 51–54.

[4] C. Blundo, V. Iovino, and G. Persiano, "Predicate encryption with partial public keys," in *Proc. CANS*, 2010, pp. 298–313.

[5] J. Katz, A. Sahai, and B. Waters, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," in *Proc. Theory Appl. Cryptogr. Techn. 27th Annu. Int. Conf. Adv. Cryptol. (EUROCRYPT)*, Berlin, Germany, 2008, pp. 146–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788414.1788423

[6] H. S. Rhee, J. H. Park, W. Susilo, and D. H. Lee, "Improved searchable public key encryption with designated tester," in *Proc. ASIACCS*, 2009, pp. 376–379.

[7] T. Moataz and A. Shikfa, "Boolean symmetric searchable encryption," in *Proc. ASIACCS*, 2013, pp. 265–276.

[8] S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "How to run Turing machines on encrypted data," Cryptol. ePrint Arch., Tech. Rep. 2013/229, Santa Barbara, CA, USA, 2013. [Online]. Available: http://eprint.iacr.org/

[9] S. Rass, "Blind Turing-machines: Arbitrary private computations from group homomorphic encryption," *Int. J. Adv. Comput. Sci. Appl.*, vol. 4, no. 11, pp. 47–56, 2013.

[10] S. Rass and P. Schartner, "System enabling licensed processing of encrypted information," U.S. Patent 14 189 234 9, Apr. 20, 2016.

[11] X. Wang, S. D. Gordon, A. McIntosh, and J. Katz, "Secure computation of MIPS machine code," Cryptol. ePrint Arch., International Association for Cryptologic Research (IACR), Nevada, NV, USA, Tech. Rep. 2015/547, 2015. [Online]. Available: http://eprint.iacr.org/

[12] X. S. Wang, C. Liu, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, May 2015, pp. 359–376. [Online]. Available: http://www.cs.umd.edu/~elaine/docs/oblivm.pdf

[13] N. G. Tsoutsos and M. Maniatakos, "HEROIC: Homomorphically EncRypted one instruction computer," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2014, pp. 1–6.

[14] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *Proc. 36th IEEE Symp. Secur. Privacy*, Oakland, CA, USA, May 2015, pp. 411–428.

[15] C. Aguilar-Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey, "Recent advances in homomorphic encryption: A possible future for signal processing in the encrypted domain," *IEEE Signal Process. Mag.*, vol. 30, no. 2, pp. 108–117, Mar. 2013.

[16] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," Cryptol. ePrint Arch., International Association for Cryptologic Research (IACR), Tech. Rep. 2014/988, Nevada, NV, USA, 2014. [Online]. Available: http://eprint.iacr.org/

[17] Y.-C. Chen, S. S. M. Chow, K.-M. Chung, R. W. F. Lai, W.-K. Lin, and H.-S. Zhou, "Computation-trace indistinguishability obfuscation and its applications," Cryptol. ePrint Arch., Tech. Rep. 2015/406, Nevada, USA 2015. [Online]. Available: http://eprint.iacr.org/

[18] S. Zahur and D. Evans, "Obliv-C: A language for extensible data-oblivious computation," Cryptol. ePrint Arch., Tech. Rep. 2015/1153, 2015. [Online]. Available: http://eprint.iacr.org/

[19] M. Hirt, "Multi-party computation: Efficient protocols, general adversaries, and voting," Ph.D. dissertation, ETH Zürich, Zürich, Switzerland, 2001.

[20] Q. Tang, "Public key encryption supporting plaintext equality test and user-specified authorization," *Secur. Commun. Netw.*, vol. 5, no. 12, pp. 1351–1362, 2012. [Online]. Available: http://dx.doi.org/10.1002/sec.418

[21] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2009.

[22] ARM Keil Microcontroller Tools. (2016). *8051 Instruction Set Manual: Opcodes*, accessed on Jul. 13, 2016. [Online]. Available: http://www.keil.com/support/man/docs/is51/is51_opcodes.htm

[23] K. Lejska. (2009). *X86 Opcode and Instruction Reference*, accessed on Jul. 21, 2016. [Online]. Available: http://ref.x86asm.net/

[24] ARM. (2016). *ARM Processor Architecture*, accessed on Jul. 21, 2016. [Online]. Available: http://www.arm.com/products/processors/instruction-set-architectures/index.php

[25] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[26] S. Zahur and D. Evans, "Poster: Obliv-C: A fast, lightweight language for garbled circuits," in *Proc. 36th IEEE Symp. Secur. Privacy*, May 2015, pp. 1–2.

[27] P. Agten, B. Jacobs, and F. Piessens, "Sound modular verification of C code executing in an unverified context," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang. (POPL)*, New York, NY, USA, 2015, pp. 581–594. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676972

[28] L. Scheffer. (2016). *Programming in Malbolge*, accessed on Jul. 13, 2016. [Online]. Available: http://www.lscheffer.com/malbolge.shtml

[29] C. LeDoux, M. Sharkey, B. Primeaux, and C. Miles, "Instruction embedding for improved obfuscation," in *Proc. 50th Annu. Southeast Regional Conf. (ACM-SE)*, New York, NY, USA, 2012, pp. 130–135. [Online]. Available: http://doi.acm.org/10.1145/2184512.2184543

[30] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[31] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. 24th USENIX Secur. Symp. (USENIX Security)*, Washington, DC, USA, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[32] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Proc. 22nd Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2015, pp. 1–15.

[33] R. Ostrovsky and W. E. I. Skeith, III, "A survey of single-database private information retrieval: Techniques and applications," in *Public Key Cryptography—PKC*, LNCS. 4450. Berlin, Germany: Springer, 2007, pp. 393–411.

[34] S. Rass, P. Schartner, and M. Wamser, "Oblivious lookup tables," in *Proc. 15th Central Eur. Conf. Cryptol. (CECC)*, 2015, pp. 8–10. [Online]. Available: http://arxiv.org/abs/1505.00605

STEFAN RASS received the master's degrees in mathematics and computer science from the Alpen-Adria-Universität Klagenfurt (AAU) in 2005, the Ph.D. degree in mathematics in 2009 and the habilitation in applied computer science and system security in 2014. He is currently an Associate Professor with the AAU, where he is involved in theoretical computer science, complexity theory, security, and cryptography. He authored numerous papers related to security and applied statistics and decision theory in security. He has co-authored the book *Cryptography for Security and Privacy in Cloud Computing*, published by Artech House. He participated in various nationally and internationally funded research projects. His research interests include applied system security and complexity theory, statistics, decision theory, and game-theory.

PETER SCHARTNER received the master's degree in telematics from the Technical University of Graz in 1997 with a focus on information security and the Ph.D. degree in computer science from the Alpen-Adria-Universität Klagenfurt in 2001 with a focus on security tokens. He participated in various nationally and internationally funded research projects. He is currently an Associate Professor with the System Security Research Group, Alpen-Adria-Universität Klagenfurt, where he is involved in theoretical computer science, algorithms and data structures, security, and cryptography. He is also a Lecturer with the Trier University of Applied Sciences. His research interests include applied system security, key management, security infrastructures, and applications for security tokens, especially smartcards.