Memory-Safe Elimination of Side Channels

Luigi Soares
Department of Computer Science
UFMG, Brazil
luigi.domenico@dcc.ufmg.br

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG, Brazil
fernando@dcc.ufmg.br

***Abstract*—A program is said to be isochronous if its running time does not depend on classified information. The programming languages literature contains much work that transforms programs to ensure isochronicity. The current state-of-the-art approach is a code transformation technique due to Wu *et al.*, published in 2018. That technique has an important virtue: it ensures that the transformed program runs exactly the same set of operations, regardless of inputs. However, in this paper we demonstrate that it has also a shortcoming: it might add out-of-bounds memory accesses into programs that were originally memory sound. From this observation, we show how to deliver the same runtime guarantees that Wu *et al.* provide, in a memory-safe way. In addition to being safer, our LLVM-based implementation is more efficient than its original inspiration, achieving shorter repairing times, and producing code that is smaller and faster.**

***Index Terms*—Side Channel, Information Leak, Program Transformation**

I. Introduction

A program is said to be *isochronous* if it always executes the same operations and accesses the same memory addresses, regardless of the input that it receives. Time invariance is a desirable property in cryptographic libraries, because its absence might be the source of side channels in these implementations [1]–[5]. Therefore, there exists much research aiming at proving that a program is time invariant (or time variant) [6]–[10], and at transforming a program to mitigate timing-based leakages [11]–[17]. Results in this area have been fundamental to uncover side channels in widely used cryptographic routines [18]–[20], and have led to the release of tools that found industrial applications in this field.

The current state-of-the-art code transformation to make a program isochronous is due to Wu *et al.* [21]. This technique has been materialized into a tool, `SC-Eliminator`, which, once applied onto a program P, produces a new program P' that is semantically equivalent to P, with two additional properties. P' executes the same instructions and accesses the same cache lines, regardless of its input. Wu *et al.* have shown that their technique is effective in eliminating side channels, both in theory and in practice, via simulations using `gem5` [22].

The Issue of Memory Safety. Nevertheless, Wu *et al.*'s approach suffers from one shortcoming, which can be reproduced in their publicly available artifact: the transformation might insert out-of-bounds memory accesses into code that was originally memory safe. Indeed, as we demonstrate in Section II-B via an example, it is not possible—in general—for a transformation to ensure both that a program is time invariant and is memory safe. The gist of the problem lays on the fact that the elimination of time invariance may force the execution of statements that were originally guarded against invalid memory accesses. Such is typically the case in algorithms that receive input arrays of unknown sizes.

A Compromise. The contribution of this paper is a suite of static code transformation techniques that transform programs to ensure their time invariance. Given the impossibility mentioned in Section II-B, we settle for a compromise. In Section II-C, we identify a class of programs that can be safely transformed, so that no invalid memory access is possible in the resulting code. This family of programs is rather small. Thus, to make *isochronification* more useful, Section III-C proposes a compromising solution. In addition to modifying the body of a function f, we also modify its interface; thus, producing a new function f_s. Said modification establishes a contract with f_s's caller by associating a symbolic bound with each input array. This contract provides the following guarantees regardless of the inputs that f_s receives:

1) f_s executes always the same instructions, meaning that it fetches always the same addresses in the instruction memory;
2) within the newly created symbolic bounds, f_s accesses always the same data, meaning that it fetches always the same addresses in the data memory;
3) outside the symbolic bounds, f_s is still memory safe (as long as f is memory safe); however, property (2) is no longer ensured.

We emphasize that the combination of properties (1) and (2) provide stronger guarantees than previous work. To this effect, we recognize a second limitation, not only in Wu *et al.*'s method, but in other approaches that try to eliminate side channels due to misses in the data cache, such as Brotzman *et al.*'s [23] recent contribution. These techniques are based on pre-loading values, to keep sensitive information directly accessible in the last-level data cache. Pre-loading is architecture dependent, for the approach is customized to the dimensions of the data cache. Our property (2) is stronger: we affirm that, once a transformation is possible, the resulting program will always access the same addresses in the data cache, independent on its sensitive inputs.

Summary of Results. A formalization of the techniques discussed in this paper have been prototyped in Haskell. A

practical version of this prototype has been implemented in LLVM. Section IV presents an evaluation of this tool in 24 cryptographic routines taken from `CTBench` [24] and Wu *et al.*'s ACM artifact [25]. Experiments in Section IV-A indicate that the transformations discussed in this paper run in time linear on the number of instructions in the target program. Section IV-B shows that the transformed code is 55% slower, on average, than the original program. Additionally, Section IV-C shows that the transformed programs are, on average, 154% larger than their original versions. In contrast, Wu *et al.*'s artifact works only for 20 of the 24 benchmarks, and leads to code that is 127% slower and 331% larger.

II. Overview

This section introduces two properties that are desirable in implementations of cryptography: *operation invariance* and *data invariance*. The first property refers to the *instruction cache*; the second, to the *data cache*. The instruction cache stores the instructions, i.e., operations, that constitute the binary encoding of a program. The data cache stores the data that said instructions manipulate in the course of their execution. We state these two properties as follows:

Property 1 (Operation Invariance)**.** A program is said to be operation invariant if it reads the same *sequence* of addresses in the instruction cache, regardless of its inputs.

Property 2 (Data Invariance)**.** A program is said to be data invariant if it reads and/or writes the same *sequence* of addresses in the data cache, regardless of its inputs.

Example 1. Function `oFdF`, in Figure 1, is not operation invariant, nor data invariant. The conditional at Line 3 depends on the contents of arrays `a` and `b`; hence, operations might vary. This conditional guards a return statement, which may prevent the program from accessing further indices of `a` and `b` in the data cache. Function `oFdT` is not operation invariant, as the conditional at Line 13 is commanded by the contents of input arrays. However, regardless of said inputs, the same elements of `a` and `b` are always accessed. Function `oTdF` shows the opposite behavior: positions accessed in the data cache depend on input array `t`. Yet, regardless of any of its three inputs, the same operations always execute in this routine. Finally, function `oTdT` is both operation and data invariant. Notice that the outcome of operations, like the selector at Line 32, might vary without compromising neither Property 1 nor Property 2. Indeed, a program that would always produce the same outputs, independent on its inputs, would not be of much service.

```
1  int oFdF(int *a, int *b) {                          !O & !D
2    for (int i = 0; i < 2; i++) {
3      if (a[i] != b[i]) {
4        return 0;
5      }
6    }
7    return 1;
8  }
9
10 int oFdT(int *a, int *b) {                          !O & D
11   int r = 1;
12   for (int i = 0; i < 2; i++) {
13     if (a[i] != b[i]) {
14       r = 0;
15     }
16   }
17   return r;
18 }
19
20 int oTdF(int *a, int *b, int *t) {                   O & !D
21   r0 = t[a[0]] != t[b[0]];
22   r1 = t[b[1]] != t[b[1]];
23   r2 = r0 | r1;
24   r3 = r2 ? 0 : 1;
25   return r3;
26 }
27
28 int oTdT(int *a, int *b) {                           O & D
29   r0 = a[0] != b[0];
30   r1 = a[1] != b[1];
31   r2 = r0 | r1;
32   r3 = r2 ? 0 : 1;
33   return r3;
34 }
```

Fig. 1. Procedures `oFdF`, `oFdT`, and `oTdT` compare a password with a secret key. Procedure `oTdF` runs a similar algorithm, albeit the indices to be compared are provided as inputs. Each program presents a different combination of operation and data invariance.

A. Properties of Program Repair

Wu *et al.* [21] define as *program repair* a code transformation procedure that changes a program P, thus producing a new program P' that is time invariant. Their transformation ensures Property 1 (operation invariance). However, they do not guarantee Property 2 (data invariance)[1]. Program repair, as proposed by Wu *et al.*, is memory unsafe: the transformed version might access unallocated memory, even if such bad accesses do not occur in the original code for the same inputs. In this paper, we are interested in transformations that are safe. To this end, we introduce a property that characterizes them:

Property 3 (Memory-Safe Program Repair)**.** Let T be a transformation that implements program repair. If P is a program, then $T(P) = P'$ is the repaired version. T is memory safe if it does not introduce out-of-bounds memory accesses in P' that do not occur in P.

Property 3 is a relation between a transformation T and a program P, valid for any input that P receives. In other words, for any input I, Property 3 holds if: (i) $P(I) = P'(I)$; and (ii) if every memory access performed during the execution of $P(I)$ is valid, then the same is true for $P'(I)$. Notice that item (ii) does not force these memory accesses to be the same.

[1]Wu *et al.* discuss alternatives to prefetch cache lines, so to ensure that hits and misses in the data cache remain independent on program inputs.

B. Time invariance vs Memory Soundness

Data invariance (Property 2) and safe program repair are, in general, two irreconcilable properties. It is possible to conceive a program P such that, for any transformation T, where $P' = T(P)$, these three statements cannot be all true: (i) P and P' produce the same outputs, given the same inputs; (ii) P' is data invariant; and (iii) the transformation is memory safe. The next example illustrates this impossibility.

Example 2. Consider function `oFdF`, in Figure 1. This function implements the following relations: $\texttt{a}[0] \neq \texttt{b}[0] \Rightarrow 0 \wedge (\texttt{a}[0] = \texttt{b}[0] \wedge \texttt{a}[1] \neq \texttt{b}[1]) \Rightarrow 0 \wedge (\texttt{a}[0] = \texttt{b}[0] \wedge \texttt{a}[1] = \texttt{b}[1]) \Rightarrow 1$. If this program receives two arrays of size one, such that $\texttt{a}[0] \neq \texttt{b}[0]$, then it returns immediately with answer 0. This input implies that any access to `a[1]` or `b[1]` would be invalid. However, there are inputs that force the access of these two memory cells. Thus, data invariance cannot characterize any semantically equivalent version of `oFdF`.

Some explanations concerning Example 2 are in order. The C programming language is inherently memory unsafe: array indexing, for instance, is not guarded against out-of-bounds accesses. Therefore, the baseline program used in Example 2 might incur in invalid memory accesses: it suffices to feed function `oFdF`, in Figure 1, with two null pointers, for instance. Such bad inputs are of no concern to Property 3. In the context of Example 2, safe program repair refers exclusively to inputs that only cause valid memory accesses in function `oFdF`. The next example further clarifies the notion of memory-safe program repair.

Example 3. The transformation proposed by Wu *et al.* [21] transforms function `oFdF` (Fig. 1) into code that is similar to function `oTdT` (in the same figure). The latter is both data and operation invariant. However, as seen in Example 2, this transformation is not memory safe, considering the inputs $\texttt{a} = \{0\}$ and $\texttt{b} = \{1\}$. In this case, function `oTdT` runs into invalid memory accesses at `a[1]` and `b[1]`.

C. Guarantees

The impossibility result discussed in Section II-B indicates that the deployment of effective program repair techniques asks for compromises. To explain the compromises that this paper adopts, we shall need to define two notions: *data consistency* and *contracts*. We define the former below, and explain the latter in Section II-D:

Definition 1 (Data Consistency)**.** A program is data consistent if it reads and/or writes the same *set* of addresses in the data cache, regardless of its inputs.

Data consistency is a form of weak data invariance. Whereas the latter forces the order of memory accesses to be always the same, the former imposes no constraints on this ordering. Data consistency gives us the necessary precondition to state our first covenant:

```
1  int new_oFdF(int *a, int *b, uint Na, uint Nb) {
2      bool r = a[0] == b[0];
3      for (uint i = 1; i < min(2, min(Na, Nb)); i++) {
4          r &= a[i] == b[i];
5      }
6      if (!r) {
7        return 0;
8      }
9      for (int j = min(Na, Nb); j < 2; j++) {
10         if (a[j] != b[j]) {
11             return 0;
12         }
13     }
14     return 1;
15 }
```

Fig. 2. Version of Function `oFdF` (Fig. 1) with extra parameters inserted to implement the memory contract.

Covenant 1. Let P be a data-consistent program and T the transformation discussed in this paper. The following guarantees hold:

1) T is memory safe;
2) $T(P)$ is operation invariant;
3) $T(P)$ is data invariant.

If P is not data consistent, then items (1) and (2) hold, but (3) does not.

D. Compromises

The techniques to be discussed in Section III meet the guarantees stated in the Covenant 1. Thus, full isochronicity is only established for data-consistent programs. To extend such guarantees to other programs, we shall adopt a compromise. To explain this compromise, we shall rely on the notion of a *memory contract*—a concept that we define as follows:

Definition 2 (Memory Contract)**.** Let $f(\ldots, a, n, \ldots)$ be a function with at least two arguments, such that a is an array, and n is an integer. The memory contract formed by the triple (f, a, n) is a precondition stating that whenever f is invoked, the array a contains at least n valid cells.

As we explain in Section III-C, we modify the signature of *data-inconsistent* functions with new *contracts*. These modifications make the functions data consistent whenever the preconditions of the contracts are met. To create memory contracts, as we shall explain in Section III, we augment the interface of a function, adding an integer to it, for each pointer that it contains. The next example shows the new interface that is created to function `oFdF`, earlier seen in Figure 1.

Example 4. Figure 2 shows a version of Function `oFdF`, transformed to incorporate a memory contract. The transformed function, `new_oFdF`, splits the code of its original version into two parts. The first part, from Line 3 to Line 8, is data consistent, as long as $\texttt{Na} \geq 2$ and $\texttt{Nb} \geq 2$. The second part, from Line 9 to Line 13, is not, due to the `return` statement at Line 11.

Notice that modifications such as those seen in Example 4, apply to a function, but not to the calling site where said

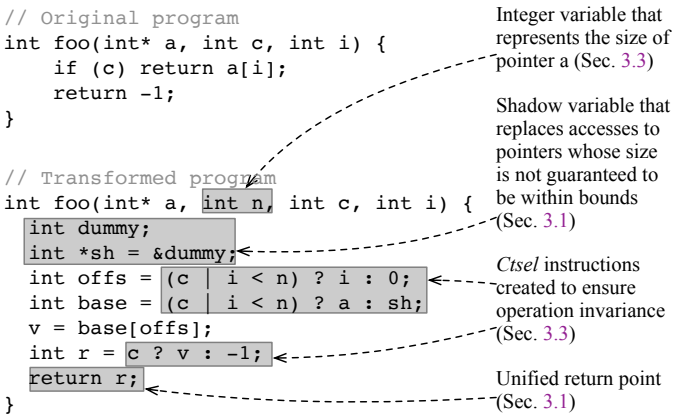

Fig. 3. Interventions carried out by the code transformation technique discussed in this paper.

function is invoked. As we describe in Section III-C2, we apply the static analysis of Paisante *et al.* [26] to determine the size of arrays. Then, we use the expressions obtained to modify that calling site. Nevertheless, state-of-the-art static analysis techniques are not, as of today, able to find symbolic bounds to every array used in a program [27]. In such cases, we assume that the length is zero, implying that data invariance is not ensured—although operation invariance and memory safety are always delivered.

III. Program Repair

This section presents the code transformation technique that we propose in this paper to bestow onto functions the properties discussed in Section II. To ease our presentation, Section III-A introduces a toy language that contains a minimum set of instructions necessary to describe our approach. Section III-B defines a suite of transformation rules that actuate on this language to produce an isochronous program. Achieving isochronicity is not always possible, as previously discussed. Thus, Section III-C explains the contract-based approach that we advocate to reach a compromise in face of such general impossibility. Finally, Section III-D shows how to extend our ideas to work on an interprocedural setting. We shall be using the program in Figure 3 to illustrate the many steps necessary to "isochronify" a program.

A. The Baseline Language

Figure 4 shows the syntax of the toy language that will be used to explain our ideas. This language contains a small, albeit Turing Complete, set of three-address code instructions which can be used to implement cryptographic routines. In Figure 4, {} denotes zero or more occurrences, [] indicates optional terms, *id* represents names of variables, n stands for numerals, and ℓ ranges over basic block labels.

Preprocessing. Henceforth, we shall assume that programs written in our toy language have four particularities. First, these programs are represented in Static Single Assignment form [28]; hence, phi-functions are part of the syntax that we adopt. Second, they contain a single return point. Third, they

$$
\begin{aligned}
\text{Program} &::= \{\ \text{BasicBlock}\ \}\\
\text{BasicBlock} &::= [\ \ell\text{: }]\ \{\ \text{Instruction}\ \}\ \text{Terminator}\\
\text{Instruction} &::= \text{alloc}\ (\textit{id}, \text{Expr})\\
&\mid \text{mov}\ (\textit{id}, \text{Expr})\\
&\mid \text{load}\ (\textit{id}, \textit{id}, \text{Value})\\
&\mid \text{store}\ (\text{Value}, \textit{id}, \text{Value})\\
&\mid \text{phi}\ (\textit{id}, \text{Value} : \ell\ \{\ , \text{Value} : \ell\ \})\\
&\mid \text{ctsel}\ (\text{id}, \text{Value}, \text{Value}, \text{Value})\\
\text{Terminator} &::= \text{jmp}\ (\ell)\\
&\mid \text{br}\ (\text{Value}, \ell, \ell)\\
&\mid \text{ret}\ (\text{Expr})\\
\text{Expr} &::= \text{Value} \mid \text{Unop Value} \mid \text{Value Binop Value}\\
\text{Value} &::= n \mid \textit{id}\\
\text{Unop} &::= - \mid\ !\ \mid \sim\\
\text{Binop} &::= + \mid - \mid * \mid\ \&\ \mid\ "|" \mid = \mid < \mid \ldots
\end{aligned}
$$

Fig. 4. Syntax of a baseline language used to implement cryptographic functions.

contain a dummy variable, called `shadow` (`sh` in the examples). The shadow variable will be used as a placeholder to memory locations of unknown size. Finally, the programs must be cycle-free. Loops, when available, must be unrolled; thus, their maximum number of iterations must be known at compilation time. Otherwise, the problem of "isochronification" would not be even well-defined [21][2]. These particularities can be ensured through a preprocessing phase.

B. Core Transformations

1) Constant-Time Selectors: Following Wu *et al.*'s notation [21], this paper also uses a special instruction *ctsel* (constant-time selector), which is architecture dependent. The operation $\texttt{ctsel}(x, c, v_t, v_f)$ assigns v_t to x, if c is true; otherwise, it assigns v_f to x. Mainstream computer architectures, such as ARM, provide direct implementations of this instruction. In x86, it can be implemented as a combination of selection and data movement.

Example 5 (Ctsel)**.** In our baseline language, $\texttt{ctsel}(x, c, v_t, v_f)$ can be implemented as $\{\texttt{mov}(c_f, c-1), \texttt{mov}(c_t, \sim c_f), \texttt{mov}(x_t, c_t\ \&\ v_t), \texttt{mov}(x_f, c_f\ \&\ v_f), \texttt{mov}(x, x_t \mid x_f)\}$.

Gathering Conditions. A constant-time selector such as $\texttt{ctsel}(x, c, v_t, v_f)$ is parameterized by a condition c. This value controls the assignment to x, i.e., the variable that the *ctsel* defines. Therefore, to build these selectors, we need to map basic blocks to the predicates that control them. To this end, we define the concept of a *path condition* as follows:

[2]We emphasize that we can still deal with programs containing unbounded loops, as long as these loops are not controlled by sensitive data.

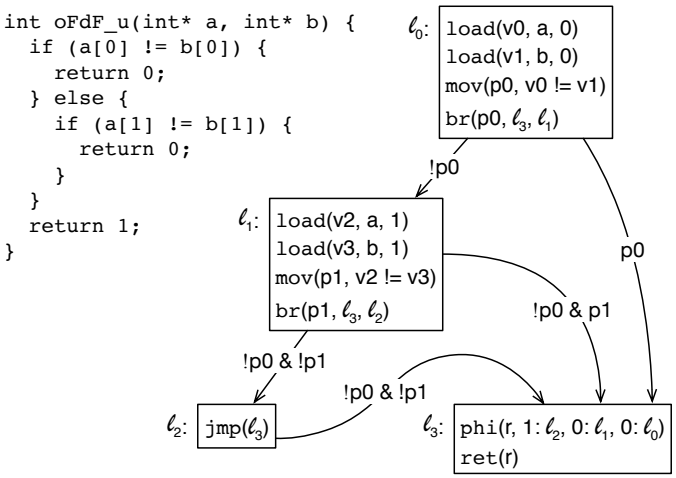

Fig. 5. Path conditions associated with each basic block in the program.

$$
\begin{array}{l}
[\texttt{Incoming}] \\
\texttt{In}[\ell] = \{\ \texttt{inc}(\ell, \texttt{terminator}(\ell_p)) : \ell_p \in \texttt{predecessors}(\ell)\ \} \\
\texttt{where}\ \ \texttt{inc}(\ell, \texttt{br}(p, \ell, \_)) = \texttt{Out}[\ell]\ \&\ p \\
\qquad\quad \texttt{inc}(\ell, \texttt{br}(p, \_, \ell)) = \texttt{Out}[\ell]\ \&\ !p \\
\qquad\quad \texttt{inc}(\ell, \_) = \texttt{Out}[\ell] \\
\\
[\texttt{Outgoing}] \\
\texttt{Out}[\ell] = c_0 \mid ... \mid c_n,\ c_i \in \texttt{In}[\ell]
\end{array}
$$

Fig. 6. Data-flow analysis that finds the path conditions that control the execution of basic blocks in a control flow graph.

Definition 3 (Path Condition)**.** A path condition $c_{0 \to n}$ is a predicate like $p_i\ \&\ ...\ \&\ p_k$, $0 \leq i \leq k \leq n$, that, once true, forces the execution of the basic block labeled by ℓ_n. Each p_i is the predicate that controls one of the branches that exists in the path from the beginning of the program until ℓ_n.

Example 6. Figure 5 shows the function `oFdF`, with its loop unrolled. The instruction $\texttt{jmp}(\ell_3)$ at ℓ_2 runs when *p0* and *p1* are false. Thus, the expression $!p0\ \&\ !p1$ is a path condition controlling the execution of that operation.

The data-flow analysis in Figure 6 collects the conditions that control each basic block in a program. This analysis classifies path conditions into *incoming* or *outgoing*. For simplicity, in the remaining of this paper we may omit the word *path* when referring to them. A basic block has a variable number of incoming conditions: one for each predecessor. In contrast, it has only one outgoing condition, which is the conjunction of all its incoming conditions. The block is in the execution path whenever the outgoing condition is true.

The uniqueness of an outgoing condition is crucial for an efficient implementation of the said data-flow analysis. First, we produce a map from labels to newly created variables that shall store the outgoing condition of each block. Abusing the notation in Figure 6, we name this map as `Out`. Since outgoing conditions are unique, `Out` will never be modified post construction. Hence, once $\texttt{In}[\ell]$ is computed, it will also never change. Therefore, a pre-order traversal of the control flow graph suffices to gather all the incoming and outgoing conditions that arise from a well-formed SSA-form program. We say that an SSA-form program is well-formed if the definition of a variable dominates all its uses. Moreover, the data-flow analysis is guaranteed to terminate, and it runs in linear time on the number of basic blocks.

Example 7. Figure 5 shows the incoming conditions associated with each basic block of `oFdF`, as computed by the analysis in Figure 6.

2) Transformation Rules: Figure 7 defines the rewriting rules that we use to implement program repair. These transformations are modeled by the following relations:

$$(i, \ell) \xrightarrow{i} (I, V) \tag{1}$$

$$(t, \ell) \xrightarrow{f} t' \tag{2}$$

Relation (1) maps $i \in$ Instruction at label ℓ to a new set of operations I, plus a set V of fresh variable names. Names in V are defined by instructions in I. The rules $\texttt{phi}_1$, $\texttt{phi}_2$ and $\texttt{phi}_n$ in Figure 7 deal with phi-functions having 1, 2 or n ($n > 2$) arguments, respectively. Rule $\texttt{phi}_1$ converts a phi-node into a `mov` operation, while the rules $\texttt{phi}_2$ and $\texttt{phi}_n$ use a map of incoming conditions (`In`) to transform phi-nodes into `ctsel` instructions. We use the notation $\texttt{In}[\ell \mapsto \{c_1, \ldots, c_k\}]$ to indicate that the entry associated with ℓ in table `In` has been updated to $\{c_1, \ldots, c_k\}$. The `store` and `load` rules require the map of outgoing conditions (`Out`), which indicates whether control flows to ℓ or not.

These two rules use a map $\mathcal{L}$ between arrays and their sizes. This map is necessary to guarantee data invariance (when possible) and is the subject of Section III-C. These rules also use a shadow memory region `sh` to ensure memory safety. Notice that the use of `sh` by the `load` rule is explicit. The address of the shadow memory might flow into the variable z_3, which denotes the position dereferenced by the `load`. On the other hand, the `store` rule uses `sh` indirectly through the variable z_3. From that, three possible cases can be derived:

- The `store` instruction should not be executed (i.e., the path condition is false) and the index used is not within safe bounds. In this case, a `load` from `sh` is followed by a `store` of the loaded value to `sh`. Thus, the operation takes no effect.
- The `store` instruction should not be executed (i.e., the path condition is false), but the index used lays inside safe bounds. Unlike the first situation, the auxiliary `load` instruction will access the original address and get the current value stored there. Then, the `store` will use this value to update the original address. Again, the operation takes not effect.

$$[\texttt{phi}_1] \quad (\texttt{phi}(x, v_0{:}\ \ell_0), \_) \xrightarrow{i} (\{\texttt{mov}(x, v_0)\}, \emptyset)$$

$$[\texttt{phi}_2] \quad \frac{\texttt{In}[\ell] = \{c_0, c_1\}}{\texttt{In} \vdash (\texttt{phi}(x, v_0{:}\ \ell_0, v_1{:}\ \ell_1), \ell) \xrightarrow{i} (\{\texttt{ctsel}(x, c_0, v_0, v_1)\}, \emptyset)}$$

$$[\texttt{phi}_n] \quad \frac{\begin{array}{c}\texttt{In}[\ell] = \{c_0, c_1, ..., c_k\}, \quad \texttt{In}[\ell \mapsto \{c_1, ..., c_k\}] = \texttt{In}', \\ \texttt{In}' \vdash (\texttt{phi}(z, v_1{:}\ \ell_1, ..., v_k{:}\ \ell_k), \ell) \xrightarrow{i} (I, V), \\ I \cup \{\texttt{ctsel}(x, c_0, v_0, z)\} = I'\end{array}}{\texttt{In} \vdash (\texttt{phi}(x, v_0{:}\ \ell_0, v_1{:}\ \ell_1, ..., v_k{:}\ \ell_k), \ell) \xrightarrow{i} (I', V \cup \{\, z \,\})}$$

$$[\texttt{load}] \quad \frac{\begin{array}{c}\texttt{Out}[\ell] = c, \quad \mathcal{L}[m] = n, \\ \{\texttt{mov}(z_0, \texttt{idx} < n), \texttt{mov}(z_1, c \mid z_0), \texttt{ctsel}(z_2, z_1, \texttt{idx}, 0) \\ \texttt{ctsel}(z_3, z_1, m, \texttt{sh}), \texttt{load}(x, z_3, z_2)\} = I\end{array}}{\texttt{Out}, \mathcal{L} \vdash (\texttt{load}(x, m, \texttt{idx}), \ell) \xrightarrow{i} (I, \{z_0, ..., z_3\})}$$

$$[\texttt{store}] \quad \frac{\begin{array}{c}\texttt{Out}, \mathcal{L} \vdash (\texttt{load}(z_4, m, \texttt{idx}), \ell) \xrightarrow{i} (I, \{z_0, ..., z_3\}), \\ \texttt{Out}[\ell] = c, \quad \{\texttt{ctsel}(z_5, c, v, z_4), \texttt{store}(z_5, z_3, z_2)\} = I'\end{array}}{\texttt{Out}, \mathcal{L} \vdash (\texttt{store}(v, m, \texttt{idx}), \ell) \xrightarrow{i} (I \cup I', \{z_0, ..., z_5\})}$$

$$[\texttt{br}] \quad (\texttt{br}(p, \ell_t, \ell_f, \ell') \xrightarrow{f} \texttt{jmp}(\ell')$$

Fig. 7. Some transformation rules used in program repair. The superscripts i and f are defined by Relations (1) and (2) in Page 5.

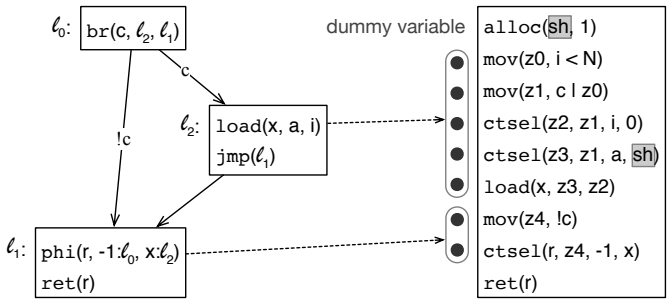

Fig. 8. The program on the left is the low-level representation of function `foo` seen in Figure 3. The program on the right is the isochronous version of that code, produced by the rewriting rules seen in Figure 7.

- The `store` instruction should be executed (i.e., the path condition is true). In this case, the original operation is performed without any modifications, updating the memory address with the new value.

Relation (2) maps $t \in$ Terminator to another $t' \in$ Terminator. The rule `br` replaces a conditional by an unconditional statement. In this case, ℓ' labels the basic block that succeeds the one that contains the `br` operation, in topological order. Figure 8 illustrates the effects of the said rules.

Theorem 1 (Correctness)**.** The transformations in Figure 7 preserve semantics.

Theorem 2 (Operation Invariance)**.** The transformations in Figure 7 yield an operation-invariant program.

Theorem 3 (Data Invariance)**.** The transformations in Figure 7 ensure data invariance when applied onto a data-consistent program.

Corollary 1 (Isochronicity)**.** Let P be a data-consistent program. The transformations in Figure 7 produce an isochronous version of P.

C. Binding Contracts

As we have discussed in Section II, the transformation proposed in this paper establishes memory contracts (Definition 2) between functions and their callers. If the contract's preconditions are met, then the transformed code is operation and data invariant, and the transformation is memory safe. Otherwise, data invariance is not guaranteed. The creation of a contract involves three actions: change of interface, modification of calling sites, and insertion of a shadow memory into the transformed function.

1) Change of interface: The creation of a contract implies a change in the function signature. We augment the function's interface with an integer for each pointer that it receives. Thus, given a function $f(\ldots, T * a, \ldots)$, we produce a new function $f'(\ldots, T\ *a, \texttt{int}\ n, \ldots)$. Following the condition stated in Definition 2, the contract expects the following relation: whenever f' is invoked, the size of the array a of type T is at least n.

2) Modification of calling sites: When applied onto whole programs, our transformation changes calling sites where modified functions are invoked. This modification involves adding expressions denoting the size of arrays as extra arguments of function calls. We use the static analysis of Paisante *et al.* [26] to infer the length of arrays. Paisante *et al.* have proposed a forward-must analysis that binds pointers to symbolic expressions denoting their sizes. To obtain an inter-procedural analysis (thus extending the intra-procedural work of Paisante *et al.*), we rely on the knowledge that the function argument following each pointer represents that pointer's maximum offset. We use this observation to propagate information across functions.

Imprecisions. This analysis only works for top-level pointers (pointers with a representation in the SSA-form program). Thus, we are not able to track the size of pointers to pointers. In this case, we assume that sizes are zero. Therefore, whenever we cannot find a symbolic estimate of the size of a pointer at a given calling site, we set its contract to zero. When this event happens, we still deliver operation invariance and memory safety. However, we cannot ensure data invariance, because different inputs might cause variations in the number of times that the shadow memory is accessed.

3) Shadow Memory: The shadow variable works as a placeholder for memory locations whose access cannot be guaranteed to be within bounds by the contract. There exists one shadow variable per function. Its size equals the size of the largest addressable word in the architecture. Figure 9 shows the rewriting procedure that adds the contract as a precondition to memory accesses. Let p be the original path condition that

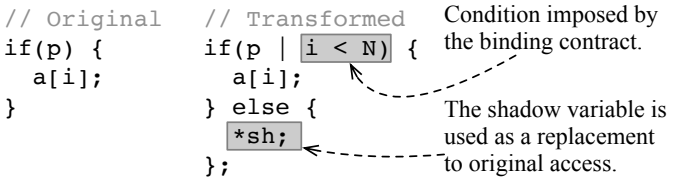

Fig. 9. Rewriting procedure that replaces potentially unsafe memory accesses with accesses to the shadow variable.

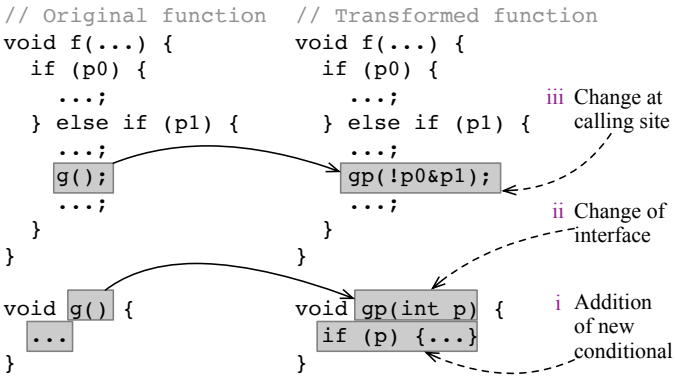

Fig. 10. Interprocedural transformations.

guards an access $a[i]$ within function f, and let (f, a, n) be its contract. The transformation guards this access with a new condition $i < n$. The new guard is safe, by Definition 2. When this condition is not met, some memory access must still happen—to ensure operation invariance. In this case, the shadow memory is accessed instead. This expedient ensures memory safety, i.e., the absence of out-of-bounds accesses, as Theorem 4 formalizes.

Example 8. Figure 8 shows how the shadow variable is used. Accesses to it have been highlighted.

Theorem 4 (Memory Safety)**.** Let f be a function. The transformations in Figure 7 are memory safe, as long as the preconditions in f's contract (Definition 2) are met.

D. Interprocedural Analysis

Cryptographic algorithms might be composed by a combination of functions. When performing the transformations that we advocate in this paper to eliminate side channels, inlining them is not an option. The combination of unrolling plus inlining might result in code that is large enough to render our techniques impractical, as Example 9 illustrates.

Example 9 (Inlining)**.** The benchmark `curve25519-donnabad.c`, distributed with `dudect`[3] contains 7,398 instructions, after full unrolling. Once functions are inlined, this number jumps up to 3,398,816. This expansion represents a growth of 460x.

Figure 10 shows the rewriting principle that we use to avoid inlining when carrying out isochronification. Every function that is called (the callee) within transformed code (the caller) is modified in three ways. First, the signature of the callee is augmented to receive a condition (Fig. 10-i). Second, the body of the callee is surrounded by a conditional test, guarded by the new condition (Fig. 10-i). Third, the caller is modified, so that the path condition at the invocation point is passed to the callee (Fig. 10-i). Notice that the conditions are computed as part of the transformation itself. In other words, the transformation just described receives, for free, the path conditions as a byproduct of the analysis discussed in Section III-B1 and shown in Figure 6.

[3]https://github.com/oreparaz/dudect

IV. Evaluation

This section evaluates the ideas presented in this paper. To this end, we analyze the following research questions:

- **RQ1:** What is the time taken to repair programs?
- **RQ2:** What is the time overhead that our program repair technique adds onto programs?
- **RQ3:** What is the size overhead that our program repair technique adds onto programs?

Software: we have implemented our techniques in LLVM version 10.0.0. To give the reader some perspective on our numbers, every experiment also reports results obtained with the version of `SC-Eliminator` (Wu *et al.*'s tool) publicly available as an ACM artifact.

Hardware: every experiment discussed in this paper has been performed on an Intel Core i5 with a clock of 2.5 GHz, and 8 GB of RAM (DDR4) operating at 2.133 MT/s. The operating system used was Manjaro Linux version 20.0.3.

Benchmarks: To answer the research questions, we use a synthetic benchmark—function `oFdF` (Fig. 1, pg. 2)—plus actual cryptographic routines. The latter is formed by `CTBench` [24] and a subset of the benchmarks distributed together with `SC-Eliminator` [25]. `SC-Eliminator` does not terminate successfully when given the three benchmarks from `CTBench`. Furthermore, it produces incorrect code when applied onto both `loki91` and `oFdF`. Thus, when discussing absolute and average numbers related to running time, we omit these five benchmarks. Nevertheless, charts seen in this Section still contain them, as we can handle them correctly.

In this paper, we assume that every input used in the cryptographic routines is sensitive. Thus, the isochronous version of each of these routines should be data and operation invariant for every input. It is possible to use tools like `FlowTracker` [18] to separate sensitive from innocuous inputs; however, neither our transformation nor `SC-Eliminator` resort to this expedient.

Timing Methodology: When reporting the time to repair programs, we run each experiment 50 times, while for the repaired code we run 1,000 times. In both cases, we eliminate outliers using their z-score with a threshold of three. The elimination of outliers is necessary, because our benchmarks run for a very short time (microseconds); hence, small running-time variations might account for large differences in final

results. Furthermore, we present results obtained from two versions: non-optimized and optimized via `opt -O1`[4].

Validation: We have used `cachegrind` to certify that all repaired programs meet the terms stated in Covenant 1. `Valgrind` verified that operation invariance and memory safety holds for all the programs with all the tested inputs. This verification pass is applied onto the transformed programs and onto their optimized versions. Data invariance is ensured for 12 benchmarks. The remaining 12 cannot be made data invariant: 11 of them are inherently data inconsistent because they use inputs to index memory[5]. For the other program, our static analysis has not been able to find symbolic expressions for arrays. It is possible to ensure data invariance by providing these bounds manually.

A. RQ1 – Repairing Time

This section reports the time that our implementation and `SC-Eliminator` take to repair programs. Both techniques are implemented as LLVM passes. We report only the time to do program repair. The rest of LLVM's processing time—the same for both implementations—is not considered.

a) Cryptographic routines: Figure 11 plots the time that our implementation and `SC-Eliminator` take to repair actual cryptographic routines. Considering only the 21 benchmarks in which `SC-Eliminator` works, our transformation required 7.159 seconds to be applied (sum of all the times observed for all the benchmarks), while `SC-Eliminator` took 56.366. Thus, the approach introduced in this paper is 7.873x faster. On average (arithmetic mean), we take 0.341 seconds per benchmark, while `SC-Eliminator` takes 2.684.

b) Empirical Asymptotic Behavior: Figure 12 compares our implementation with `SC-Eliminator` on the `oFdF` routine[6] (Fig. 1), considering different sizes for the input arrays. For arrays of size `N`, the main loop is modified—statically—to run `N` iterations. This experiment lets us probe the asymptotic behavior of both tools, by varying the size of the program that must be generated. Both implementations seem to be linear on `N`. If we let C_t represent the time that our implementation takes, and C_m the time that Wu *et al*'s implementation takes to run, then we have almost perfect fits for the lines $C_t = 0.0002 \times \texttt{N} - 0.0313$ and $C_m = 0.001 \times \texttt{N} - 0.215$ (both have $R^2 > 0.94$ with p-values close to zero). Our implementation scales better, given the smaller slope.

[4]We are showing results for `-O1` only, to save space, but our findings remain true for the other levels that we have evaluated, namely -O2, -O3, and -Oz.

[5]There exist also programs that are inherently operation variant. Such is the case of a program that contains a loop bounded by sensitive data. The benchmarks used in this paper do not show this behavior.

[6]We have observed that the code that `SC-Eliminator` produces for `oFdF` is not correct. Nevertheless, we show the time that `SC-Eliminator` takes to produce this program, to give the reader some perspective on the quality of our implementation.

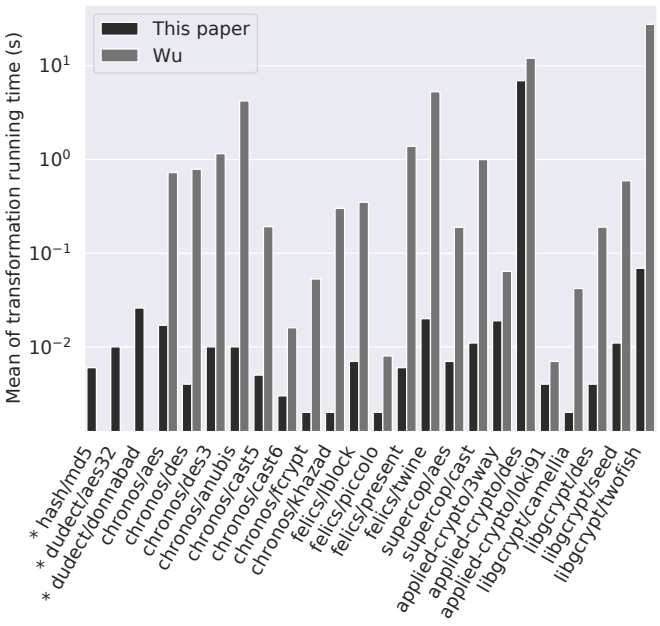

Fig. 11. Time to apply program repair in actual implementations of cryptography. Benchmarks prefixed by an asterisk are those in which Wu *et al.*'s artifact failed to run.

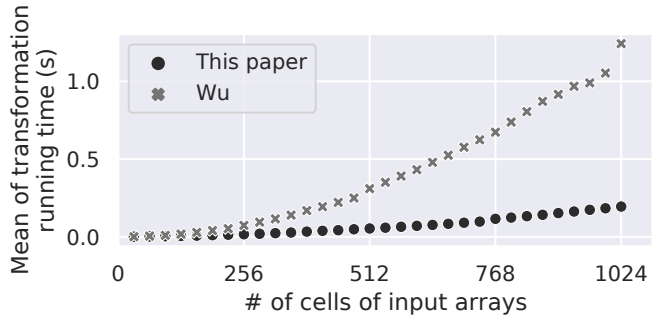

Fig. 12. Time to repair function `oFdF`, considering different sizes of the input arrays `a` and `b`. This size determines the maximum number of iterations in the loop.

B. RQ2 – Time Overhead

Repair slows down programs as a consequence of extra instructions inserted to preserve time invariance. This section analyzes such impact.

a) Cryptographic routines: Figure 13 shows the impact of program repair on cryptographic benchmarks, considering codes without and with optimizations. Regarding only the code that `SC-Eliminator` handles, our technique slows down programs by 55%, on average (geometric mean of ratios). `SC-Eliminator` causes a slowdown of 127%. Optimizations reduce this impact: our slowdown is of 50%, whereas Wu's is 106%. The averaged running time of all the optimized benchmarks is 77.0 μs. Same number for the programs after our transformation is 127.4 μs. Wu's is 266.001 μs. Notice that the implementation of `SC-Eliminator` differs from ours in a number of ways, which cause the relative slowdown

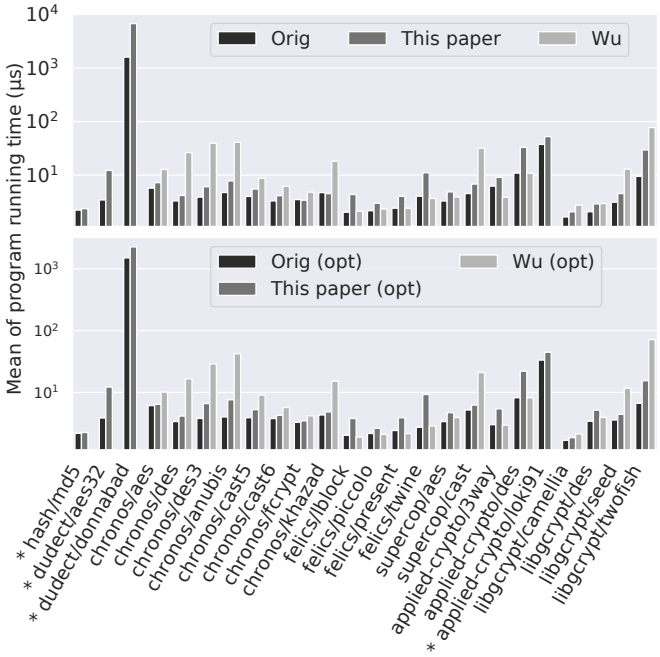

Fig. 13. Impact of program repair on the running time of implementations of cryptography. Benchmarks prefixed by an asterisk are those that Wu *et al.*'s program repair either failed to transform or produced incorrect results.

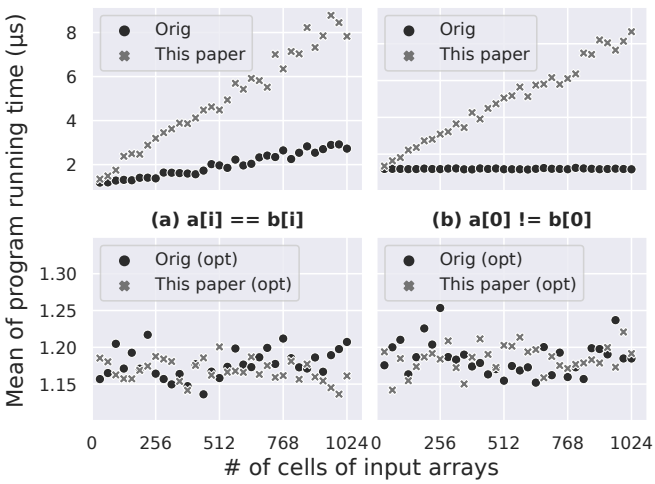

Fig. 14. Comparison of the running time of different implementations of function `oFdF` (Fig. 1, pg. 2). This size determines the maximum number of iterations in the loop.

of the code that it produces. Mostly, it inserts code to read arrays before loops, to bring the data to cache. In our case, this procedure is not necessary. Also, Wu et. al's approach requires that conditional statements be transformed into Single-Entry-Single-Exit regions—such transformation accounts for a handful of instructions that are absent in the code that we produce.

b) Empirical Asymptotic Behavior: Figure 14 compares the running time of original and transformed versions of function `oFdF` for different sizes of input arrays. As seen in Section IV-A, the length of these arrays determines the number of iterations of the loop at Lines 2-6 in Fig. 1. Regardless of the contents of the array, the transformed function will run the same operations. Visual comparison of Figures 14 (a) and (b) provides some indication of this fact. The original function, in contrast, only runs all the iterations of the main loop when the two input arrays store the same values. Both programs show linear behavior on `N`, the number of cells in the input array. Let T_o be the running time of the original program, and let T_t be the running time of the transformed version, when both receive arrays of equal contents. The following relation yields a strong linear fit: $T_t = 3.8\mathtt{T_o} - 2.52$, with $R^2 > 0.94$, and p-value close to zero. Thus, our transformation, in the absence of compiler optimizations, imposes a a four-fold overhead onto the original program.

However, compiler optimizations have a much stronger effect onto the transformed program. The lower charts in Figure 14 support this statement. The total absolute running time that the original (optimized) program takes to process the 32 arrays (with equal contents) is 37.5 seconds. The transformed program, in turn, once optimized, takes 37.2 seconds in total to process the same 32 arrays. In this case, it does not matter if the arrays are the same or not: the algorithm, even when optimized, is still time invariant.

C. RQ3 – Size Overhead

Repair increases programs, because it augments them with the extra instructions necessary to ensure time and data invariance. This section analyzes this growth.

a) Cryptographic routines: Figure 15 compares the size—in number of LLVM instructions—of original and repaired cryptographic libraries. There are two versions of repaired code: ours and the ones produced by `SC-Eliminator`. When analyzing non-optimized programs, on average, our technique increases code size by 154% (geometric mean of ratios). `SC-Eliminator` increases code size by 331%. In absolute numbers, the original programs add up to 141,945 LLVM instructions. Our transformation moves this number up to 427,145 instructions, while `SC-Eliminator` yields 786,235 instructions. Optimizations have a strong effect on this growth. The optimized versions of the original programs contain 89,326 instructions. Programs repaired with our techniques, when optimized, add up to 150,782 instructions; `SC-Eliminator` yields 661,735 instructions.

b) Empirical Asymptotic Behavior: Figure 16 analyzes the asymptotic behavior of our transformation, considering unoptimized and optimized repaired versions of `oFdF` (Fig. 1). The figure uses log-scale; however, growth, in both cases, is linearly proportional to the number of iterations of the loop at Lines 2-6 of Figure 1. For the unoptimized codes, the linear relation is essentially a perfect fit ($R^2 = 1$). Once optimizations are considered, the coefficient of determination is much weaker: 0.26 (comparing optimized versions of the original and transformed codes). In general, the transformed program will be 3.8 times larger than the original program (without

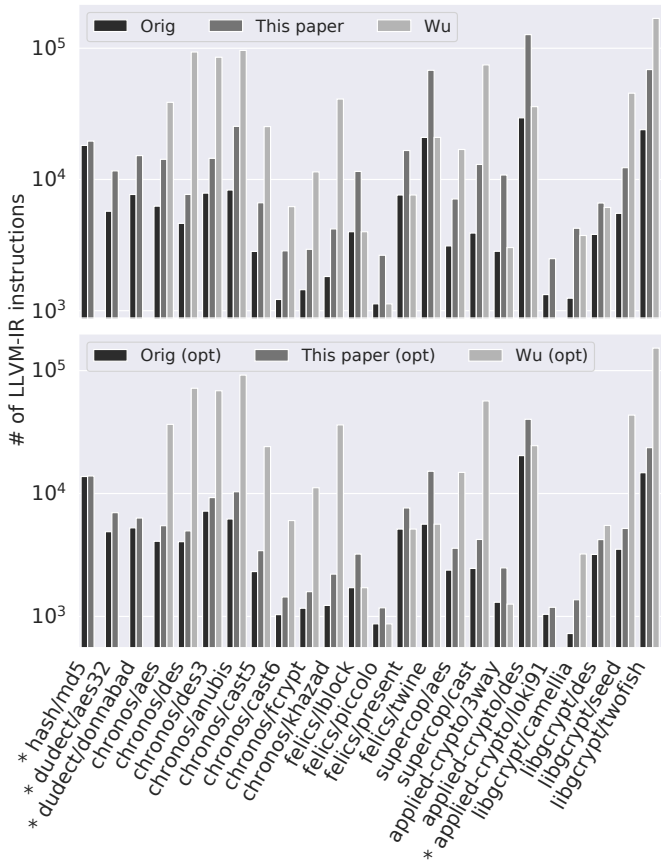

Fig. 15. Impact of program repair onto the size of actual implementations of cryptography. Benchmarks prefixed by an asterisk are those that Wu *et al.*'s program repair either failed to transform or produced incorrect results.

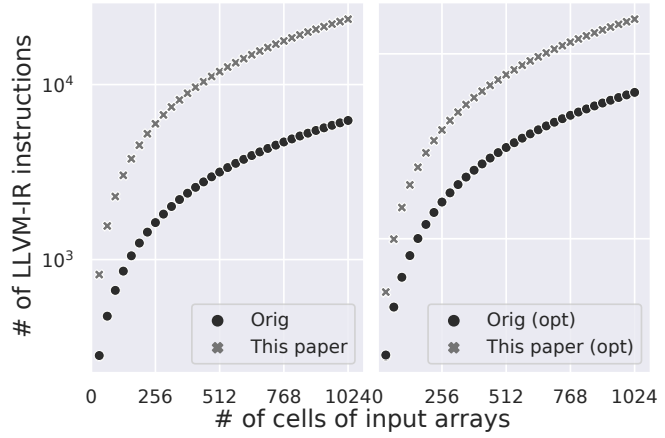

Fig. 16. Comparison of the number of LLVM instructions in different implementations of function `oFdF` (Fig. 1, pg. 2).

optimizations) and 1.8 times larger when optimizations are considered. But, in this last case, ratios vary substantially.

V. Related Work

Information leakage is a critical problem in cryptography, with several vulnerabilities having been uncovered in the past three decades. Early forays in the field started with Kocher [4], who, in the mid-nineties, demonstrated timing attacks in algorithms such as Diffie-Hellman and RSA. Since then, flaws have been exposed in widely used cryptographic routines [1]–[3], [5]. For instance, Brumley and Tuveri [3] presented a timing attack vulnerability in OpenSSL's ladder implementation for elliptic curves over binary fields. Consequently, much effort has been put into developing techniques to either detect, mitigate or completely eliminate side channels. This paper fits into the third category.

a) Detection of Side Channels: Most of the literature concerning side channels refer to their identification, not to their elimination. For instance, Rodrigues *et al.* [18] rely on the properties of Static Single Assignment form to design an efficient detection technique that operates on the LLVM intermediate representation. Around the same time (2016-17), Almeida *et al.* [7] and Reparaz *et al.* [6] introduced techniques to determine whether code runs in constant time or not. Ngo *et al.* [8] described a type system for verifying that a code correctly implements constant-resource behavior. More recently, Guarnieri *et al.* [10] introduced a framework for specifying hardware-software contracts that assert which program executions an adversary can distinguish. A CPU satisfies a contract if, whenever two program executions agree on all observations, they are guaranteed to be indistinguishable by the adversary at the microarchitectural level.

b) Elimination of Side Channels: Compiler optimizations may break constant-time properties that hold at the source code level. From this observation, Barthe *et al.* [9] have presented a modified version of `CompCert` [29] that preserves such guarantees. Yet, Barthe's work is rather preventing the need for program repair than actually repairing code. On the other hand, there exists a wide range of approaches to mitigate information leakage due to time-based side channels [8], [11]–[15], [21], [30]. The seminal work in the field is due to Johan Agat, who has proposed a type system and a type-directed transformation to repair programs [11]. Agat's technique, and also several of its successors, work by equalizing the time spent on distinct branches within a program. These approaches essentially seek a trade-off between the overhead imposed upon the transformed program and the amount of leakage that they mitigate. For instance, Niari *et al.* explicitly guarantee a user-specified maximum acceptable performance overhead [15].

However, as stated by Wu *et al.*, such methods deliver weak guarantees, due to the presence of hidden states at microarchitectural levels and related performance optimizations inside modern CPUs. Thus, to the best of our knowledge, the strongest guarantees in terms of program repair are currently provided by Wu *et al.*'s [21]. We believe that Rane *et al.*'s [30] work might provide equally strong guarantees as Wu's, yet such guarantees are not explicitly stated. Nevertheless, as we have discussed in Section II, these approaches still suffer from a number of shortcomings that justify the developments in this paper. In particular, although both Wu *et al.* and Rane *et al.* provide arguments about the absence of side channels,

they do not discuss the issue of memory safety—a problem present in both techniques. As a final remark about security guarantees, notice that all the works discussed in this section, ours included, can only change the program, but not the micro-instructions that execute it. Sensitive inputs can still influence the order in which micro-instructions run, if the target processor speculates on values stored in registers.

VI. Conclusion

This paper presented a code transformation technique to eliminate time-based side channels. This transformation ensures that a program always runs the same operations and accesses the same data, regardless of inputs. Contrary to previous work, our program repair is memory safe: it will not cause the modified program to access out-of-bounds memory. To make safety possible, we augment the interface of functions with memory contracts: extra arguments representing valid limits of arrays. In addition to enabling memory safety, said contracts lead to a simple and efficient interprocedural implementation of program repair. This implementation is today publicly available as an LLVM pass. The empirical evaluation discussed in this paper indicates that it outperforms previous techniques in terms of repairing time and quality of the generated code.

Acknowledgment

This work has been made possible by grants from different research agencies, namely CNPq, CAPES and FAPEMIG. We thank Augusto Noronha and José Wesley Magalhães for reading a draft of this work. We also thank the CGO reviewers, for all the time and expertise they have put into our paper.

References

[1] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *Smart Card Research and Applications*, J.-J. Quisquater and B. Schneier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 167–182.

[2] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701 – 716, 2005, web Security. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128605000125

[3] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security – ESORICS 2011*, V. Atluri and C. Diaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 355–371.

[4] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer, 1996, pp. 104–113.

[5] W. Schindler, "A timing attack against rsa with the chinese remainder theorem," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç. K. Koç and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 109–124.

[6] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *DATE*. Leuven, BEL: European Design and Automation Association, 2017, p. 1701–1706.

[7] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 53–70.

[8] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann, "Verifying and synthesizing constant-resource implementations with types," in *Security and Privacy*. Washington, DC, USA: IEEE, 2017, pp. 710–728.

[9] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371075

[10] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," 2020.

[11] J. Agat, "Transforming out timing leaks," in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 40–53. [Online]. Available: https://doi.org/10.1145/325694.325702

[12] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan. 2012. [Online]. Available: https://doi.org/10.1145/2086696.2086702

[13] A. Fell, H. T. Pham, and S.-K. Lam, "Tad: Time side-channel attack defense of obfuscated source code," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 58–63. [Online]. Available: https://doi.org/10.1145/3287624.3287694

[14] J. Van Cleemput, B. De Sutter, and K. De Bosschere, "Adaptive compiler strategies for mitigating timing side channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 35–49, 2020.

[15] S. Tizpaz-Niari, P. Černý, and A. Trivedi, "Quantitative mitigation of timing side channels," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 140–160.

[16] S. Chattopadhyay and A. Roychoudhury, "Symbolic verification of cache side-channel freedom," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2812–2823, 2018.

[17] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *2020 IEEE Symposium on Security and Privacy (SP)*. New York, NY, USA: IEEE, 2020, pp. 965–982.

[18] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 110–120. [Online]. Available: https://doi.org/10.1145/2892208.2892230

[19] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leakage in cache attacks via symbolic execution," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 1, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3288758

[20] T. Basu, K. Aggarwal, C. Wang, and S. Chattopadhyay, "An exploration of effective fuzzing for side-channel cache leakage," *Software Testing, Verification and Reliability*, vol. 30, no. 1, p. e1718, 2020, e1718 stvr.1718. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1718

[21] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*. New York, NY, USA: Association for Computing Machinery, 2018, p. 15–26.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, 2011.

[23] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation," in *Security & Privacy*. New York, NY, USA: IEEE, 2019, pp. 505–521.

[24] A. C. Lopes and D. F. Aranha, "Benchmarking tools for verification of constant-time execution," in *SBSEG*. Bento Goncalves, Brazil: SBC, 2017, pp. 716–726, https://github.com/arthurlopes/ctbench.

[25] M. Wu, S. Guo, P. Schaumont, and C. Wang, "[ISSTA '18 Artifact Evaluation] Eliminating Timing Side-Channel Leaks using Program Repair," ACM, Jun. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1299357

[26] V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira, "Symbolic range analysis of pointers," in *CGO*. New York, NY, USA: Association for Computing Machinery, 2016, p. 171–181.

[27] M. Rodrigues, B. Guimaraes, and F. M. Q. Pereira, "Generation of in-bounds inputs for arrays in memory-unsafe languages," in *CGO*. Washington, DC, USA: IEEE Press, 2019, p. 136–148.

[28] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *POPL*. New York, NY, USA: Association for Computing Machinery, 1989, p. 25–35.

[29] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, p. 107–115, 2009.
[30] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *SEC*. USA: USENIX Association, 2015, p. 431–446.

APPENDIX A
PROOFS

This section contains proofs of theorems that we have omitted from the main body of the paper, due to space constraints. Figure 17 shows rules that were omitted from Figure 7. These rules are not essential to the understanding of our technique; however, they are necessary for proving properties of our code transformation. In particular, Figure 17 extends Figure 7 with the rules necessary to rewrite a program, in addition to individual instructions. These rules follow closely, albeit with different syntax, a Haskell implementation that we have used to test our ideas. Rewriting is given in the format:

$$\texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle \{i\} \cup P, \ell, P' \rangle \xrightarrow{p} \langle P, \ell', P' \cup I \rangle \tag{3}$$

In and *Out* are the maps of incoming and outgoing conditions produced after running the analysis in Figure 6. $\mathcal{L}$ is a map between pointers and their lengths, as discussed in Section III. P is the original program that must be transformed, ℓ is the label of the current basic block being analyzed, and P' is the transformed version of P, at the time some instruction i is processed. The initial configuration is always a triple formed by the original program, the label of its entry point and the empty set. Notice that, for simplicity, we represent the transformed program P' as a set. We trust the reader to understand that once we add a new set of instructions I to P', an event denoted by $P' \cup I$, these new instructions will be inserted into P' in an ordering that respects data dependencies. Since we assume that P contains a unique exit point (as discussed in Section III-A), the last rule applied is always *[exit]*. The result of this rule is a triple formed by the empty set, a special value ϵ indicating that there are no remaining basic blocks, and a set of instructions $P' \cup \{ret(e)\}$ that constitutes the transformed program. Rule [*trans*] defines a transitive closure. The relation $\xrightarrow[*]{p}$ indicates that the right-hand side is obtained from a finite sequence of applications.

a) A Note on Semantics Preservation: The first theorem that we discuss in this section, Theorem 1, states that our transformations preserve program semantics. This statement must be understood with care. The original and the transformed codes are different: the latter contains a number of variables that are not present in its original version. We argue that set of variables defined during an arbitrary execution of the original program is a subset of the ones defined during any execution of its isochronous version. More formally, let $\texttt{defs} : \texttt{Program} \times \texttt{Input} \rightarrow \texttt{Var}$ be the set of definitions of a program, given an specific input. Additionally, let P and P' be, respectively, the original and transformed codes. Then, for any input I, it follows that:

$$[\texttt{alloc}] \quad (\texttt{alloc}(x, e), \_) \xrightarrow{i} (\{\texttt{alloc}(x, e), \emptyset)$$

$$[\texttt{mov}] \quad (\texttt{mov}(x, e), \_) \xrightarrow{i} (\{\texttt{mov}(x, e)\}, \emptyset)$$

$$[\texttt{ctsel}] \quad (\texttt{ctsel}(x, c, v_t, v_f), \ell) \xrightarrow{i} (\{\texttt{ctsel}(x, c, v_t, v_f), \emptyset)$$

$$[\texttt{jmp}] \quad (\texttt{jmp}(\ell), \_) \xrightarrow{f} \texttt{jmp}(\ell)$$

$$[\texttt{inst}] \quad \frac{i \in \text{Instruction}, \quad \texttt{In}, \texttt{Out}, \mathcal{L} \vdash (i, \ell) \xrightarrow{i} (I, \_)}{\texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle \{i\} \cup P, \ell, P' \rangle \xrightarrow{p} \langle P, \ell, P' \cup I \rangle}$$

$$[\texttt{flow}] \quad \frac{t \in \{\texttt{br}, \texttt{jmp}\}, \quad \texttt{next}(\ell) = \ell', \quad (t, \ell') \xrightarrow{f} t'}{\texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle \{t\} \cup P, \ell, P' \rangle \xrightarrow{p} \langle P, \ell', P' \cup \{t'\} \rangle}$$

$$[\texttt{exit}] \quad \texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle \{\texttt{ret}(e)\}, \ell, P' \} \rangle \xrightarrow[*]{p} \langle \emptyset, \epsilon, P' \cup \{\texttt{ret}(e)\} \rangle$$

$$[\texttt{trans}] \quad \frac{\begin{array}{c}\texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle \{i\} \cup P, \ell, P' \rangle \xrightarrow{p} \langle P, \ell', P'' \rangle, \\ \texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle P, \ell', P'' \rangle \xrightarrow[*]{p} \langle \emptyset, \epsilon, P''' \rangle\end{array}}{\texttt{In}, \texttt{Out}, \mathcal{L} \vdash \langle \{i\} \cup P, \ell, P' \rangle \xrightarrow[*]{p} \langle \emptyset, \epsilon, P''' \rangle}$$

Fig. 17. The complement of the transformation rules seen in Figure 7. We assume that the exit point is unique, i.e. the program contains only one *ret*, which is the last instruction. The set *Instruction* is the same as defined in Figure 4. Furthermore, $next(\ell)$ gives the label of the basic block that succeeds ℓ in topological order.

$$\texttt{defs}(P, I) \subseteq \texttt{defs}(P', I) \tag{4}$$

Notice that $\texttt{defs}(P', I)$ is the same for every I, since P' is operation invariant (Theorem 2). Moreover, $\texttt{defs}(P', I)$ encompasses all the possible definitions from the original program, along with the definitions produced by each rule in Figure 7. Hence, given an input I, we can understand the correctness stated by Theorem 1 as the preservation of the states of the variables in the set $\texttt{defs}(P, I)$, during the execution of $P'(I)$.

Let us assume that the original program contains an instruction such as $\texttt{load}(x, m, idx)$. This operation might not run in P, due its path condition being false. However, it may execute in P', due to the contract created to ensure data consistency. To understand why such divergence happens, and its consequences, we shall use the examples in Figure 18.

Example 10. Figure 18 shows a function (labeled "Original") that receives a vector as an argument, plus a condition `Z` that determines when the vector will be read. Its "Augmented" version contains a new parameter, `N_v`, which sets a memory contract, i.e., the triple (`foo0`, `v`, `N_v`). This variable is used as an extra condition that enables access of `v` at Line 11. This access would not happen in the original program. Thus,

```
1  foo0(int* v, int i, int x, int Z) {
2    x = 0;
3    if (Z) {
4      x = v[i];
5    }
6    return x;
7  }                                          Original
8
9  foo0(int* v, int N_v, int i, int x, int Z) {
10   x = 0;
11   if (Z | (i < N_v)) {
12     x = v[i];
13   }
14   return x;
15 }                                          Augmented
16
17 foo0(int* v, int i, int x, int Z) {
18   x0 = 0;
19   if (Z) {
20     x1 = v[i];
21   }
22   // ret = phi(x1, x0)
23   return Z ? x1 : x0;
24 }                                          Original-SSA
25
26 foo0(int* v, int N_v, int i, int x, int Z) {
27   x0 = 0;
28   if (Z | (i < N_v)) {
29     x1 = v[i];
30   }
31   return Z ? x1 : x0;
32 }                                          Augmented-SSA
```

Fig. 18. From top to bottom: a program, its version augmented with a contract, a high-level representation of the original program in SSA form, using a selector instead of a phi-function, and the same representation of the transformed program.

variable x ends up assigned in the Augmented routine, whereas it is not assigned after initialization in the original program. However, as we will show in the proof of Theorem 1, this new definition of x bears no impact on the semantics of function foo0. In other words, its value cannot be neither stored in memory, nor returned from foo0. It is important to remember that the program will be converted to SSA form, and there will be phi-functions selecting only definitions whose path condition is true in the original program. Continuing with our example, Figure 18 shows the original program in SSA form, and its augmented version, also in SSA form. Notice that the selector at Line 31 will only let the value of x1 escape from foo0 if the condition Z is true. The second condition, $\texttt{i} < \texttt{N\_v}$, is only used to guard loads and stores, and cannot interfere on how the arguments of phi-functions are selected.

Theorem 1 [Correctness]. The transformations in Figure 7 (viz. Figure 17) preserve semantics.

> **Proof:** The proof is a case analysis on each one of the rewriting rules seen in Figures 7 and 17, plus induction on rule [*trans*] (Fig. 17). This case analysis requires the semantics of the baseline language. However, instead of providing this semantics, we shall present the reasoning in informal English:

alloc Nothing changes; hence, semantics is preserved. Same reasoning works for rules [*alloc*], [*mov*], [*ctsel*], [*jmp*], and [*exit*].

$\texttt{phi}_1$ A single-arity phi-function is equivalent to a move instruction, so the transformation is trivially correct.

$\texttt{phi}_2$ A two-arity phi-function of the form $\texttt{phi}(x, v_0{:}\ \ell_0, v_1{:}\ \ell_1)$, at a block labeled by ℓ, is transformed into $\texttt{ctsel}(x, c_0, v_0, v_1)$, where c_0 and c_1 are the incoming path conditions that reach ℓ from ℓ_0 and ℓ_1, respectively. Given that the execution flowed to ℓ, either c_0 or (exclusively) c_1 must be true. Hence, if c_0 holds, *ctsel* selects the value v_0; otherwise, it selects v_1. In both cases, it chooses the operand associated with the basic block that preceded ℓ in the execution path, thereby preserving the semantics of the original operation.

$\texttt{phi}_n$ the proof follows by induction on the number of arguments of the phi-function. We assume that the transformation preserves semantics for k arguments. Therefore, I is a correct reconstruction of $\texttt{phi}(z, v_1{:}\ \ell_1, \ldots, v_k{:}\ \ell_k)$. We conclude by observing that the instruction $\texttt{ctsel}(x, c_0, v_0, z)$ will assign v_0 to x only when c_0 is true. Otherwise, it assigns z to x, which, from the inductive hypothesis, is a correct implementation of the other k arguments of the phi-function.

load a $\texttt{load}(x, m, idx)$, at a basic block ℓ, is transformed into a sequence of instructions I such that the last one is $\texttt{load}(x, z_3, z_2)$. The variables z_2 and z_3 are the result of evaluating two *ctsels*. The first is responsible for selecting the correct index, while the second selects the memory region that shall be accessed. Assuming the program is in SSA-form, there are three ways for a load to affect its result: (i) as an operand of a *store*; (ii) as a value selected by some *phi-node*; and (iii) as the expression returned by the program. Since we assume that the program contains a unique *ret* operation, the case (iii) is safe. The exit block is always visited; hence, the original load shall be executed. For case (ii), we have already proved that the three rules related to phi-functions preserve semantics; thus, the load only takes effect if ℓ would be in the execution path of the original program. Finally, for situation (iii) we shall see that stores also preserve semantics. In essence, a store instruction at ℓ' only modifies the value of some memory location if $\texttt{Out}[\ell']$ is true. In this case, the original store is performed. We refer the reader to Example 10 for more details.

store a $\texttt{store}(v, m, idx)$, at a basic block ℓ, is transformed into a set of instructions I such that the last one is $\texttt{store}(z_5, z_3, z_2)$. As in the rule [*load*], the variables z_2 and z_3 contains, respectively, the index and the memory region that shall be accessed by the store operation. The variable z_5 is the result of evaluating a $\texttt{ctsel}(z_5, c, v, z_4)$, where c is the outgoing condition of ℓ and z_4 is the current value stored at the memory location being accessed. Therefore, if c is true, the original instruction is executed; otherwise, the operation stores z_4, i.e. the old value, which means that nothing changes. Hence, the semantics is preserved.

br Effectively, this rule eliminates a conditional branch $br(p, \ell_t, \ell_f)$ at ℓ, linking ℓ with the basic block that succeeds it in the topological order of the control flow graph. Let I_t and I_f be disjoint sets of instructions within the two distinct paths created by the *br* operation. In other words, either the instructions in I_t or (exclusively) in I_f are executed. Since, from the previous cases, we know that the discussed operations preserve the semantics of a program, an instruction $i_t \in I_t$ only takes effect if and only if p is true; for $i_f \in I_f$, is the opposite case, i.e. if and only if p is false. Hence, we conclude that the semantics is preserved.

flow The proof follows immediately from the cases for rules [*jmp*] and [*br*].

trans The proof follows by induction on the number of steps in the relation $\xrightarrow[*]{p}$. □

Theorem 2 [Operation Invariance]. The transformations in Figure 7 (viz. Figure 17) yield an operation-invariant program.

Proof: Let

$$\langle P, \ell, P' \rangle \xrightarrow[*]{p} \langle \emptyset, \epsilon, P'' \rangle,$$

be a finite sequence of rule applications, producing the final program P''. Let k be the total number of conditional branches within the original program P. In other words, there are $k \geq 0$ transformations of the form

$$(\texttt{br}(p, \ell_t, \ell_f), \ell') \xrightarrow{f} \texttt{jmp}(\ell')$$

in the sequence above. Each of them replaces one of the n conditionals by an unconditional statement. Therefore, the transformed code P'' contains no instructions of the form $br(p, \ell_f, \ell_t)$, meaning that all operations within P'' shall be executed regardless of any input. Hence, P'' is operation invariant. □

Theorem 3. The transformations in Figure 7 ensure data invariance when applied onto a data-consistent program.

Proof: A data-consistent program P always access the same set of memory addresses, regardless of its inputs. As stated in Section II-D, data consistency is a form of weak data invariance that does not impose any constraints in the ordering of the accesses. In other words, P always access the same set of memory locations for any possible execution path. Let k be the number of paths in the execution flow of P, and n be the number of memory-related instructions in each one of them. Moreover, Let

$$\langle P, \ell, P' \rangle \xrightarrow[*]{p} \langle \emptyset, \epsilon, P'' \rangle,$$

be a finite sequence of rule applications, producing the final program P''. Note that the rule [*br*] eliminates every conditional statement, thereby generating a transformed code with a unique execution path. Furthermore, P'' contains $k \times n$ memory-related operations, which, for all intents, are the original n instructions within each one of the k execution paths of P. Therefore, P'' always access the same set of memory addresses, in the same order. Hence, it is data invariant. □

Corollary 1 [Isochronicity]. Let P be a data-consistent program. The transformations in Figure 7 produce an isochronous version of P.

Direct consequence of Theorems 2 and 3. □

b) A Note on Data Consistency.: Theorem 3 delivers data invariance only if the input program is data consistent. This pre-condition—data consistency—might seem, in principle, too restrictive. However, in the absence of data consistency, memory-safe program repair is impossible in general, as illustrated by Examples 2 and 3. There exist a general family of programs that cannot be made data consistent. These are programs in which the input itself is used to index memory, as Example 11 illustrates.

Example 11 (Data-Inconsistent Program)**.** The code labeled "Original", in Figure 18 is not data consistent. The address accessed at Line 4 depends on the input. The three other versions of that program remain data inconsistent, as each access to `v[i]` is indexed by `i`, which is a program input.

As Example 11 shows, a memory contract will not guarantee data consistency if inputs are used to index memory—it is part of the semantics of the algorithm to be data inconsistent. Rather, the *raison d'être* of a memory contract is to ensure memory safety. Notice that reading the entire memory at the beginning of the target function—an approach often used to ensure data invariance—does not bring isochronicity. First, because this approach depends on the size of the cache, i.e., it is architecture dependent. Second, because it is always possible to devise an input array larger than any cache; therefore, misses will invariably happen.

In the absence of data consistency, we still provide operation invariance and memory safety (Covenant 1). Consequently, the results in Corollary 1 are true for data-consistent programs. However, the Theorem 4, to be discussed, holds for any program. Finally, a last remark is in order: memory contracts are meant to be established automatically; however, there are situations in which such automatic bindings are not possible, e.g., when dealing with pointers of pointers, for instance. In such situations, developers can still use repaired functions. Nevertheless, they will have to furnish sizes of arrays themselves in order to meet the requirements imposed by the extended function signatures.

Theorem 4. Let f be a function. The transformations in Figure 7 are memory-safe, as long as the preconditions in f's contract (Definition 2) are met.

Proof: Let

$$\langle P, \ell, P' \rangle \xrightarrow[*]{p} \langle \emptyset, \epsilon, P'' \rangle,$$

be a finite sequence of rule applications, producing the final program P''. P is the set of instructions in function f. There are two memory-related instructions in the baseline language depicted in Figure 4: *load* and *store*. Let k be the number of loads within the original program P. In other words, there are $k \geq 0$ transformations of the form

$$(\texttt{load}(x, m, idx), \ell) \xrightarrow{i} (I, V)$$

in the sequence above. The instruction

$$\texttt{load}(x, z_3, z_2) \in I$$

is the last one in the sequence produced by the rule [*load*]. The variables z_2 and z_3 are defined by two *ctsel* operations

that use the same condition z_1. In turn, $z_1 = c \mid z_0$, where c is the outgoing condition of the basic block ℓ that contains the load instruction, $z_0 = idx < n$, and n is the symbolic bound associated with the pointer m. If c is true, the original instruction is executed; thus, the address accessed is the same as for the original program (i.e. $z_3 = m$ and $z_2 = idx$). Otherwise, the instruction would not be in the execution path of the original code, but it still need to be executed in the isochronous version; we call it a *zombie* operation. Let $idx > n$. This implies that m at position idx cannot be used by the *zombie* load (i.e. it is not safe). In such case, the location given by z_3 at position z_2 is of the *shadow* memory, which still is a valid region. The same reasoning applies to *store* operations. Therefore, we conclude that every memory related instruction in P'' is memory-safe, as long as the preconditions in f's contract are met. □

APPENDIX B
ARTIFACT

A. Abstract

Our artifact is a docker image (Arch Linux, AMD64) containing all the scripts and binaries necessary to reproduce the experiments described in section IV. It also includes the source code of all the benchmarks that we used. Although the source code of our transformation tool is not distributed with the docker image, it is publicly available on Github: https://github.com/lac-dcc/lif/tree/artifact/cgo.

The workflow consists of building the benchmarks, collecting data, and generating the charts shown in section IV. For the first step, it is expected that our tool works for all programs, including those in which Wu et al.'s prototype failed. Then, we collect information about the execution time of the benchmarks, running time of the transformations, and the size of the original and modified programs. We also use `cachegrind` to extract data related to the read/write hits and misses, which helps to verify whether the produced codes are indeed isochronous.

B. Artifact check-list (meta-information)

- **Algorithm: Memory-safe elimination of timing-based side channels.**
- **Program: CTBench and the benchmarks distributed in Wu *et al.*'s ACM artifact.**
- **Compilation: Clang 10.0.1 and LLVM 10.0.1 (both included in the docker image).**
- **Transformations: LLVM and Wu et al.'s transformation pass (included).**
- **Binary: All required binaries included (AMD64).**
- **Run-time environment: The artifact is not OS specific and only requires Docker to be installed.**
- **Run-time state: For low-variance results, we recommend running the experiments on a system without other compute- or memory-intensive applications running in the background.**
- **Metrics: Execution time and program size.**
- **Output: Graphs, along with the data (CSV files) used to generated them (available inside the container).**
- **Experiments: Use the `run.sh` script to build the benchmarks, collect the data, and generate the graphs.**
- **How much disk space required (approximately)?: 2.2 GB.**
- **How much time is needed to prepare workflow (approximately)?: Depends solely on the time do download the (compressed) docker image (approximately 1 GB).**
- **How much time is needed to complete experiments (approximately)?: 2~3 hours.**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: GPL3.**
- **Archived (provide DOI)?: Yes (10.5281/zenodo.4266870).**

C. Description

1) How delivered: Our source code, benchmarks, and scripts are available on Github: https://github.com/lac-dcc/lif/tree/artifact/cgo. The artifact (docker image) can be downloaded either from a Docker Hub repository (https://hub.docker.com/repository/docker/luigidcsoares/lif) or from Zenodo (https://zenodo.org/record/4266870).

2) Software dependencies: In order to use the artifact, only Docker is required (the dependencies are all included). To build the project from scratch, it is necessary to have installed Clang 10.0.1 and LLVM 10.0.1. You will also need Python 3.8.6, along with a few packages, in order to reproduce the experiments. In this case, check the READMEs in the Github repository.

D. Installation

There is a ready-to-use `Docker`[7] image available, filled with all the binaries and scripts necessary to reproduce the experiments presented in this paper. In order to use it, you will have to download the image. In this case, there are two options:

- Docker Hub:

```
$ docker pull luigidcsoares/lif:cgo
```

- Zenodo: You can either download the image from https://zenodo.org/record/4266870 or use `zenodo_get`[8]:

```
$ zenodo_get 10.5281/zenodo.4266869
$ docker load -i lif-cgo.tar.gz
```

E. Experiment workflow

The experiment workflow consists of compiling a set of benchmarks in three distinct ways: (i) without any modifications; (ii) transformed by the prototype that implements the technique described in this paper; and (iii) modified by the program repair method proposed by Wu et al. Furthermore, for each one of the three compilation workflows, two versions of the same program will be produced: unoptimized and optimized via `-O1` flag.

Each benchmark contains at least two distinct *main* source codes, with different inputs. After compiling them, we run the original version as well as the code produced by Wu et al.'s and our tools, and verify if the outputs are the same. Then, we

[7] https://www.docker.com/

[8] https://github.com/dvolgyes/zenodo_get

collect the following information from each program version: read/write hits and misses (from `cachegrind`); execution time of the program; size of the program; and execution time of the transformations (both our and Wu et al.'s). Once the data is collected, we use them to plot the charts.

F. Evaluation and expected result

In order to reproduce the research questions, you will need to run docker with a volume attached (a shared folder between the host and the container). This way, you will have access to the generated figures in the host. For that, you can proceed as follows:

```
$ docker run \
   -v $(pwd)/figures:/lif/llvm/bench/figures \
   -it luigidcsoares/lif:cgo /bin/bash
```

Once inside the container, you will have access to all the scripts, binaries, and benchmarks necessary to run the experiments. After finishing them, the shared folder *figures* will be filled with all the graphs presented in section IV. First you will need to build all the benchmarks. To do that, run the following command:

```
$ ./run.sh -b
```

There will be three types of outputs: (i) *pass*, in case the original and the transformed programs produce the same result; (ii) *fail*, in case the results are different; and (iii) *LLVM error*, if the tool could not be applied to the program. As seen in section IV, Wu et al.'s tool could not handle some programs. In such cases, the output will be either (ii) or (iii). In constrast, our prototype is expected to pass on all the benchmarks.

Now, run the commands below to collect the data about each benchmark and generate the graphs. The second line will copy the generated charts to the shared folder *figures*, renaming them to match with the figures in the paper. This way, all the charts will be available in your host, in the folder `$(pwd)/figures`.

```
$ ./run.sh -c && ./run.sh -p
$ cp results/pass_time.pdf figures/11.pdf \
  && cp results/exec_time.pdf figures/13.pdf \
  && cp results/size.pdf figures/15.pdf \
  && cp comp/results/pass_time.pdf figures/12.pdf \
  && cp comp/results/exec_time.pdf figures/14.pdf \
  && cp comp/results/size.pdf figures/16.pdf
```

In addition, inside the container you will also have access to files generated individually for each benchmark, which were used to produce the charts. You will also have access to the cachegrind reports. You can visualize them using `vim`:

```
$ vim meng/chronos/aes/results/exec_time.csv
$ vim meng/chronos/aes/results/pass_time.csv
$ vim meng/chronos/aes/results/size.csv
$ vim meng/chronos/aes/results/cachegrind.csv
```

G. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging