Side-Channel Elimination via Partial Control-Flow Linearization

LUIGI SOARES, UFMG, Brazil
MICHAEL CANESCHE, UFMG, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Partial control-flow linearization is a code transformation conceived to maximize work performed in vectorized programs. In this paper, we find a new service for it. We show that partial control-flow linearization protects programs against timing attacks. This transformation is sound: given an instance of its public inputs, the partially linearized program always runs the same sequence of instructions, regardless of secret inputs. Incidentally, if the original program is publicly safe, then accesses to the data cache will be data oblivious in the transformed code. The transformation is optimal: every branch that depends on some secret data is linearized; no branch that depends on only public data is linearized. Therefore, the transformation preserves loops that depend exclusively on public information. If every branch that leaves a loop depends on secret data, then the transformed program will not terminate. Our transformation extends previous work in non-trivial ways. It handles C constructs such as "goto", "break", "switch" and "continue", which are absent in the FaCT domain-specific language (2018). Like Constantine (2021), our transformation ensures operation invariance, but without requiring profiling information. Additionally, in contrast to SC-Eliminator (2018) and Lif (2021), it handles programs containing loops whose trip count is not known at compilation time.

CCS Concepts: • **Software and its engineering → Compilers**; • **Security and privacy → Cryptography**.

Additional Key Words and Phrases: Side channel, cryptography, compiler

ACM Reference Format:
Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. 2023. Side-Channel Elimination via Partial Control-Flow Linearization. 1, 1 (March 2023), 44 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A program is said to be *isochronous* if it always executes the same instructions and accesses the same sequence of memory addresses, independent of its secret inputs. Isochronous programs do not leak time-related information [Kocher 1996]; therefore, isochronicity is an essential property in implementations of cryptographic routines [Almeida et al. 2016; Almeida et al. 2020; Barthe et al. 2019]. In view of this importance, much work has been done to detect time-variant code [Almeida et al. 2016; Barthe et al. 2019; Guarnieri et al. 2021; Ngo et al. 2017; Reparaz et al. 2017] or to remove sources of time variance [Agat 2000; Almeida et al. 2020; Borrello et al. 2021; Chattopadhyay and Roychoudhury 2018; Cleemput et al. 2012; Fell et al. 2019; Gruss et al. 2017; Tizpaz-Niari et al. 2019; Van Cleemput et al. 2020; Wu et al. 2018a]. And yet, the implementation of a static code

Authors' addresses: Luigi Soares, Computer Science, UFMG, Belo Horizonte, Minas Gerais, Brazil, luigi.domenico@dcc.ufmg.br; Michael Canesche, Computer Science, UFMG, Belo Horizonte, Minas Gerais, Brazil, michaelcanesche@dcc.ufmg.br; Fernando Magno Quintão Pereira, Computer Science, UFMG, Belo Horizonte, Minas Gerais, Brazil, pronesto@gmail.com.

XXXX-XXXX/2023/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

transformation technique able to make programs operation invariant with respect to secret inputs remains an elusive endeavor for programs containing loops.

The Breakthroughs of 2021. Current methodologies to achieve operation invariance with regards to secret inputs consist in *linearizing* the program's control-flow graph. Linearization removes branches from a program. Until recently, the state-of-the-art approach to perform linearization was due to Wu et al. [2018a]. In 2021, Soares and Pereira [2021] proposed `Lif` as an improvement of Wu et al.'s transformation, to prevent it from introducing out-of-bounds accesses into the program. The techniques of Wu et al. and Soares and Pereira are fully static: they do not require executing a program to change it. However, they cannot deal with programs containing loops, unless these loops have bounds known at compilation time, i.e. are fully unrollable. This limitation exists even if the loops only branch on public information.

Still in 2021, Borrello et al. [2021] introduced `Constantine`, a dynamic alternative to Soares and Pereira's static technique. Borrello et al. execute the program and use runtime information like memory addresses and the outcome of branches to linearize the part of the code that could be covered during the execution. Borrello et al.'s strategy handles programs with general loops. To the best of our knowledge, it does not insert invalid memory accesses into programs. However, it also has limitations. Our personal experience with `Constantine` is that it is hard to find inputs that reach particular branches that should be linearized. In this paper, we show that it is possible to handle programs with general loops with a static analysis, hence bringing the results of Soares and Pereira closer to Borrello et al.'s.

Enter Partial Control-Flow Linearization. In 2018, Moll and Hack [2018] introduced partial control-flow linearization (PCFL): a code-optimization technique to speed up programs in the *Single-Instruction, Multiple-Data* (SIMD) model [Flynn 1972]. An SIMD program is processed by multiple threads running in lockstep. The hardware fetches one instruction at a time, which is forwarded to all the threads. Thus, these threads process the same instruction simultaneously, albeit on different data. In an SIMD program, some branches can be proven to be *uniform*, meaning that they always yield the same outcome for the threads that execute them together. The other branches are called *divergent*. Moll and Hack's PCFL removes the divergent branches from the program, linearizing the blocks controlled by them. This transformation keeps the uniform branches unchanged. In principle, PCFL seems unrelated to side-channel resistance. However, replace "uniform" with *public* and "divergent" with *secret*, and we obtain a beautiful algorithm to make programs isochronous!

The Contributions of this Paper. This paper shows how to adapt Moll and Hack's partial control-flow linearization to make the sequence of instructions executed by a program invariant with regard to secret inputs that said program might receive. As we shall see in §4.7, PCFL suffices to ensure *Cryptopgrahic Constant-Time* (CCT) behavior [Barthe et al. 2021, §2.3] — or, as we call it, *isochronicity* — for *publicly-safe* programs[1]. To employ our code transformation, users must indicate which program inputs are secret. No more interventions are necessary. The generated code achieves the following properties, which guarantee standard notions [Rafnsson et al. 2017, §4] of *confidentiality* and *non-interference* [Zdancewic and Myers 2001]:

Operation Invariance: given an arbitrary instance of the public inputs, every execution of the transformed program processes the same sequence of addresses in the instruction cache.

Data Invariance: given an arbitrary instance of the public inputs, every execution of the transformed program processes the same sequence of reads and writes in the data cache

[1]Definition 2.11 (Page 8) explains the notion of public safety, which was originally formalized by Cauligi et al. [2019]. For now, it suffices to know that memory accesses in a publicly-safe program are either independent from secret inputs or happen within buffers whose bounds are known to the code linearizer.

— this property is guaranteed whenever the original program is *publicly safe* [Cauligi et al. 2019, §3.2.3].

Memory Safety: the transformed program only contains out-of-bounds memory accesses that already exist in the original program, given any input feed to it.

Termination: a loop in the transformed program only terminates due to public information. A loop controlled only by secret data will not terminate.

The last property — termination — implies that the transformation proposed in this paper might turn a terminating program into an infinite loop. Non-termination emerges when a loop can only terminate due to conditions that depend on secret information. In other words, a partially linearized loop whose function is called with public inputs that do not trigger any of the loop exits will run forever. The most trivial case is when every exit condition of the original loop depends on secret information, a scenario that can be statically determined after partial control-flow linearization. Section 4.5.3 will provide further discussion on this subject, analyzing the merits and shortcomings of partial control-flow linearization in regard to termination. Although some of the properties that we meet are the same as those delivered by Soares and Pereira, the algorithm proposed in this paper is very different, and the code that it produces is equally unrelated. Programs produced by our transformation still might have branches, as long as these branches are not influenced by secret data. Hence, we expand previous work in many ways:

(1) *Static Generality:* in contrast to previous work [Soares and Pereira 2021; Wu et al. 2018a], our transformation handles programs with loops, even if these loops cannot be fully unrolled.
(2) *Static Efficiency:* in contrast to previous static transformations [Soares and Pereira 2021; Wu et al. 2018a], we preserve branches controlled by public information, avoiding the unnecessary execution of unreachable code.
(3) *Decidability:* our transformation is fully static. Thus, in contrast to a dynamic tool like `Constantine` [Borrello et al. 2021], it does not require test cases that exercise all the program branches.
(4) *Convenience:* in contrast to a domain-specific language such as `FaCT` [Cauligi et al. 2019], programmers can write publicly-safe code directly in C and still obtain isochronicity.

Summary of Results. We have implemented our ideas in LLVM 13.0 [Lattner and Adve 2004]. Section 5 compares this implementation with `Lif`, `Constantine`, `SC-Eliminator` and `FaCT` in regard to 13 programs whose inputs can be split into public and secret data. Section 5.4 certifies that the transformed programs meet the guarantees previously enumerated, i.e. operation invariance in general, data invariance for publicly-safe programs, memory safety and termination. Figure 1 summarizes results reported in §5 for the nine benchmarks that all the tools can handle. `Lif` and `SC-Eliminator` cannot handle three benchmarks due to unbounded loops. These two tools and `Constantine` failed to produce correct output for another one. Notice that the size of the code produced by `Lif` and `SC-Eliminator` is much bigger because these tools require that loops are fully unrolled. Our prototype is available at https://github.com/lac-dcc/lif as an improvement of `Lif`, the artifact produced by Soares and Pereira [2021].

2 THREAT MODEL AND GUARANTEES

We adopt a *string-of-addresses* threat model used by previous work [Cauligi et al. 2019; Soares and Pereira 2021]. We assume an attacker who can observe the sequence of addresses accessed by a program in the instruction and data caches. In other words, the attacker has access to the trace formed by the memory addresses read or written by a program, including the address of the

Tool	Original	PCFL (*this*)	Lif	CTT-Orig	CTT-CFL	SC-Orig	SC-CFL
Size (LLVM insts)	330.78	377.00	15,342.11	464.67	365.89	10,863.56	8,444.89
Running time (us)	3.23	5.21	12.09	14.94	6.27	5.19	4.60
Lineariz. time (ms)		33.49	271.61	2,045.22	65.19	2,697.06	2,149.28

Fig. 1. Summary of results from Section 5 with regard to the nine benchmarks that all the tools can handle. Numbers are arithmetic means. Measurements happen after programs are transformed and then optimized with `LLVM opt` -O3. "LLVM instrs." refer to the number of LLVM instructions in the intermediate representation of the benchmark. "Lineariz. time" is the time taken to linearize the control flow of the programs. `Original` refers to the benchmark without any transformation. `PCFL` is our technique. `CTT` refers to `Constantine`; `SC` refers to `SC-Eliminator`. These two tools can do control- and data-flow linearization. Hence, `-Orig` refers to their original implementations, and `-CFL` refers to the implementation with only control-flow linearization. Our approach only does control-flow linearization, but achieves data invariance for publicly-safe programs.

instructions fetched during execution[2]. This model delivers stronger guarantees than the *hit-miss* model typically adopted in cache-based timing attacks. The hit-miss model considers an attacker who has access to the sequence of cache hits and misses [Borrello et al. 2021; Wu et al. 2018a; Zhang et al. 2022], assuming a *deterministic timing model* [Balliu et al. 2014, §3]. Both models, string-of-addresses and hit-miss, follow the general execution-trace model discussed by Zdancewic and Myers [2001, §2]; however, the string-of-addresses model includes fewer programs. Example 2.1 further explains these differences.

Example 2.1. Any program that contains a memory access indexed by secret information, e.g. `t[pw[i]]` in function `oTdF` (Figure 2 (c)), will yield a different string of addresses accessed in the data cache, thus leaking information. Leaks happen even if only cache hits occur in practice. In the words of Cauligi et al. [2019], these programs are not considered *publicly safe*. In fact, an indirect comparison between these two models — string-of-addresses and hit-miss — is available in the work of Zhang et al. [2022]. Zhang et al. compare two tools, `CAPE` and `Lif`. The former uses the hit-miss model; the latter, the string-of-addresses model. Out of five benchmarks transformed by `CAPE`, `Lif` could secure only one, albeit with stronger guarantees. Thus, whereas `CAPE` tries to guarantee that the string of cache lines accessed during program execution is invariant, `Lif` tries to guarantee that the string of addresses (within those lines) is invariant.

The string of addresses might contain addresses from the data cache and addresses from the instruction cache. If the addresses from the instruction cache are invariant with respect to secret inputs, then the program is said to be *operation invariant*, according to Definition 2.2:

Definition 2.2 (Operation Invariance). Let $\mathcal{I} = (\mathcal{S}, \mathcal{P})$ be the inputs taken by a program P, where $\mathcal{S}$ is the set of secret and $\mathcal{P}$ is the set of public inputs. Let τ_1 and τ_2 be traces of operations that correspond to the execution of P when given instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$. Program P is said to be operation invariant if $\tau_1 = \tau_2$ always holds for any $\mathcal{S}_1$ and $\mathcal{S}_2$, $\mathcal{S}_1 \neq \mathcal{S}_2$.

Example 2.3. Functions `oFdF` and `oFdT` depicted in Figures 2 (a) and (b) are not operation invariant, whereas `oTdF` and `oTdT` from Figures 2 (c) and (d) are. To illustrate the concept of operation invariance, consider Figures 2 (a) and (d), with public inputs `g = {0}` and `n = 2`. The traces of operations that correspond to function `oFdF` called with the secret inputs `pw = {0}` and `pw = {1}`

[2]Microarchitectural behavior, such as speculative execution, store-to-load forwarding, and prefetching influence which addresses flow to the cache. Thus, the string-of-addresses threat model is still an approximation of what an actual attacker can learn with regard to program execution.

are, respectively,

$$\tau_1^{(a)} = (\texttt{i = 0, i < n, g[i] != pw[i], i++, i < n, ret 1}) \quad \text{and}$$
$$\tau_2^{(a)} = (\texttt{i = 0, i < n, g[i] != pw[i], ret 0}).$$

Notice that the string-of-addresses threat model refers to traces of *instruction addresses*. However, for clarity, we use the instructions themselves, instead of their addresses, when presenting execution traces. This style of presentation is not ambiguous, because our examples do not contain repeated instructions. Continuing with the example, if the function oTdT is called with the same secret inputs as above — namely, pw = {0} and pw = {1} — then the corresponding traces are

$$\tau_1^{(d)} = (\texttt{r = 1, i = 0, i < n, r \&= g[i] != pw[i], i++, i < n, ret r}) \quad \text{and}$$
$$\tau_2^{(d)} = (\texttt{r = 1, i = 0, i < n, r \&= g[i] != pw[i], i++, i < n, ret r}).$$

Notice that $\tau_1^{(a)} \neq \tau_2^{(a)}$, while $\tau_1^{(d)} = \tau_2^{(d)}$. That is, the instructions executed in oFdF vary depending on the secret pw, but are always the same in function oTdT regardless of the value of the secret.

a

```c
// g (guess) is public.
// pw (password) is secret.
// n is public.
int oFdF(int *g, int *pw, int n) {
    for (int i = 0; i < n; i++)
        if (g[i] != pw[i]) return 0;
    return 1;
}
```

$\overline{O} \wedge \overline{D}$

b

```c
// g (guess) is public.
// pw (password) is secret.
// n is public.
int oFdT(int *g, int *pw, int n) {
    int r = 1;
    for (int i = 0; i < n; i++)
        if (g[i] != pw[i]) r = 0;
    return r;
}
```

$\overline{O} \wedge D$

c

```c
// g (guess) and t (table) are public.
// pw (password) is secret.
// n is public.
int oTdF(int *g, int *pw, int *t, int n) {
    int r = 0;
    for (int i = 0; i < n; i++) {
        // secret-dependent index: pw[i]
        r |= t[g[i]] != t[pw[i]];
    }
    return r ? 0 : 1;
}
```

$O \wedge \overline{D}$

d

```c
// g (guess) is public.
// pw (password) is secret.
// n is public.
int oTdT(int *g, int *pw, int n) {
    int r = 1;
    for (int i = 0; i < n; i++)
        r &= g[i] != pw[i];
    return r;
}
```

$O \wedge D$

Fig. 2. Operation and data invariance are program properties that can occur independently from each other in code. A program can be both, operation and data invariant, or present only one of these properties, or none of them. Functions oFdF, oFdT, oTdF and oTdT compare the user's guess g with a secret password pw. (a) oFdF returns immediately whenever two elements are different; as such, it is neither operation nor data invariant. (b) oFdT always reads the same array cells; hence, it is data invariant, but not operation invariant. (c) oTdF is operation invariant; however, it has indirect accesses through a table t, using secret-dependent indices (pw[i]), and thus it is not data invariant. (d) oTdT always performs the same sequence of instructions and memory accesses; thus, it is both operation and data invariant.

If the addresses from the data cache are invariant with respect to secret inputs, then the program is said to be *data invariant*. Definition 2.4 states this concept.

Definition 2.4 (Data Invariance). Let $\mathcal{I} = (\mathcal{S}, \mathcal{P})$ be the inputs taken by a program P, where $\mathcal{S}$ is the set of secret and $\mathcal{P}$ is the set of public inputs. Let τ_1 and τ_2 be traces of memory addresses in the data cache that correspond to the execution of P when given instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$, $\mathcal{S}_1 \neq \mathcal{S}_2$. Program P is said to be data invariant if $\tau_1 = \tau_2$ always holds.

Example 2.5 (Data Invariance). Consider, again, the procedures from Figure 2. Functions oFdF and oTdF are not data invariant, whereas oFdT and oTdT are. To illustrate the notion of data invariance, let us observe the execution of functions oFdF (Figure 2 (a)) and oTdT (Figure 2 (d)) when called with public inputs g = {0,0} and n = 2. When called with secret inputs pw = {0,0} and pw = {1,0}, the traces of memory addresses that correspond to function oFdF are, respectively,

$$\tau_1^{(a)} = (\texttt{g[0], pw[0], g[1], pw[1]}) \qquad \text{and}$$
$$\tau_2^{(a)} = (\texttt{g[0], pw[0]}).$$

Notice that, similarly to Example 2.3, we use the syntax of an array access, e.g. g[0], instead of the address of that memory position, to represent traces in the data cache. Continuing with the example, if the function oTdT is called with the same secret inputs as above — namely, pw = {0,0} and pw = {1,0} — then the corresponding traces are, respectively,

$$\tau_1^{(d)} = (\texttt{g[0], pw[0], g[1], pw[1]}) \qquad \text{and}$$
$$\tau_2^{(d)} = (\texttt{g[0], pw[0], g[1], pw[1]}).$$

Notice that $\tau_1^{(a)} \neq \tau_2^{(a)}$, while $\tau_1^{(d)} = \tau_2^{(d)}$. That is, the data addresses accessed in oFdF vary depending on the secret pw, but are always the same in oTdT regardless of the value of the secret.

Function oTdT is a typical constant-time transformation of function oFdF, e.g. it is close to the code produced by SC-Eliminator [Wu et al. 2018a]. However, as pointed out by Soares and Pereira [2021], the transformed version might incur into out-of-bounds memory accesses that were absent in the original program[3]. Example 2.6 illustrates the problem of memory safety.

Example 2.6 (Memory Safety). Suppose that functions oFdF and oTdT are called with arguments g = {0}, pw = {1} and n = 2. The former, oFdF, will immediately return after comparing g and pw, whereas the latter will move on to the second iteration of the loop, thus accessing arrays g and pw at index 1, which is out of bounds. These invalid accesses do not occur in the original code oFdF.

In general, it is impossible to ensure data invariance in a memory-safe way. Soares and Pereira [2021] have shown that there are programs that cannot be transformed to be data invariant and memory safe while preserving their semantics. Thus, usually code linearizers settle for a compromise: they replace potentially unsafe memory accesses with accesses to a dummy memory position — the *shadow memory*. Nevertheless, the use of a shadow memory as a surrogate address might cause the transformed code to be data variant. Example 2.7 illustrates this issue.

Example 2.7. As seen in Example 2.6, function oTdT would be a memory-unsafe linearized version of function oFdF, both seen in Figure 2. Figure 3, in turn, shows a transformation of the function oFdF that preserves memory safety. This linearization assumes that the sizes of arrays g and pw can be inferred symbolically by a static-analysis tool. These sizes are represented by integers g->size and pw->size. In-bounds memory accesses happen as is; however, out-of-bounds addresses are replaced with the dummy variable shadow. Consider that memsafe_oTdT is called with public inputs g->data = {0, 0} and n = 2. Furthermore, assume that it was not possible to infer the size of the arrays and thus g->size = pw->size = 0. Let τ_1 be the trace produced for the secret input pw->data = {0, 0} and let τ_2 be the trace produced for the secret input pw->data = {1, 0}.

[3]The implementations used by Soares and Pereira [2021] as examples can be seen as loop-free versions of the codes from Figure 2. Recall that neither Lif [Soares and Pereira 2021] nor SC-Eliminator [Wu et al. 2018a] can deal with general loops.

Then, we have

$$\tau_1 = (\texttt{g->data[0], pw->data[0], g->data[1], pw->data[1]}) \text{ and}$$
$$\tau_2 = (\texttt{g->data[0], pw->data[0], shadow, shadow}).$$

As a second example, consider that function `memsafe_oTdT` is called with arguments `g->data = {0}`, `g->size = pw->size = 1` and `n = 2`. Let τ_3 be the trace produced for the secret `pw->data = {1}` (similar to Example 2.6) and let τ_4 be the trace produced for the secret `pw->data = {0}`. Then,

$$\tau_3 = (\texttt{g->data[0], pw->data[0], shadow, shadow}) \text{ and}$$
$$\tau_4 = (\texttt{g->data[0], pw->data[0], g->data[1], pw->data[1]}).$$

Notice that, as opposed to function `oTdT`, when `pw->data = {1}` there was no out-of-bounds accesses in the second iteration of the loop (trace τ_3). Notice, also, that the out-of-bounds accesses verified in τ_4 would also happen in the original function `oFdF`, when called with arguments `g = {0}`, `pw = {0}` and `n = 2`. In other words, `memsafe_oTdT` has not introduced new out-of-bounds accesses and thus the transformation that produced `memsafe_oTdT` preserves memory safety.

```c
typedef struct ptr_int_wrapped {
    int *data;
    // the object's inferred size
    int size;
} ptr_int_wrapped;

// g (guess) is public.
// pw (password) is secret.
// n is public.
int memsafe_oTdT(ptr_int_wrapped *g, ptr_int_wrapped *pw, int n) {
    int *shadow = (int *) malloc(sizeof(int));
    int r = 1;
    // If true, indicates that the loop has not entered in "dummy" mode. By dummy mode, we
    // mean that the loop would have already exited in the original version of this code.
    int loop_cond = 1;
    for (int i = 0; i < n; i++) {
        // If "loop_cond" is true, arrays g and pw are used. The original arrays are also used
        // if"loop_cond" is false (i.e. the loop is now operating in dummy mode), but the
        // accesses are in-bounds. Otherwise, the shadow memory is used.
        int *g_addr = ctsel(loop_cond | (i < g->size), g->data, shadow);
        int *pw_addr = ctsel(loop_cond | (i < pw->size), pw->data, shadow);
        int g_idx = ctsel(loop_cond | (i < g->size), i, 0);
        int pw_idx = ctsel(loop_cond | (i < pw->size), i, 0);
        int g_i = g_addr[g_idx];
        int pw_i = pw_addr[pw_idx];
        loop_cond &= g_i == pw_i;
        r &= loop_cond;
    }
    return r;
}
```

Fig. 3. This function is a partially linearized version of the function `oFdF` seen in Figure 2 (a). The code transformation introduced in this paper transforms function `oFdF` into function `memsafe_oTdT` automatically. We named the repaired function as `memsafe_oTdT` to indicate that the code transformation corresponds to a memory-safe version of the transformation that led to function `oTdT` (Figure 2 (d)). Notice that our techniques are implemented in LLVM, and work at the level of the LLVM intermediate representation — we show code in C for readability. We assume the existence of a `ctsel` function that conditionally select between two values and runs in constant time (see Section 4.2). Function `memsafe_oTdT` is operation invariant, but is data invariant only if the array sizes, i.e. `g->size` and `pw->size`, are known to be greater than or equal to the loop bound `n`. In this paper, we omit how the signature of `oFdF` is augmented with the sizes of the objects, because our interventions are similar to the ones described in detail in Section III.C of Soares and Pereira's work.

There are alternative approaches for solving the problem of memory safety. `Constantine` [Borrello et al. 2021], for instance, "*obliviously accesses all the locations that the original program can possibly reference for any initial input*". In practice, `Constantine` surrounds every memory access `m[i]` with a loop that traverses every address in the buffer `m`. The drawback of this approach, if applied naïvely, is that the performance overhead may be prohibitive. Furthermore, `Constantine` requires the sizes of objects to be public. Example 2.8 illustrates the behavior of `Constantine`.

Example 2.8. Consider the function `oFdF` from Figure 2 (a). If we assume that the sizes of arrays `g` and `pw` are fixed and only their *contents* can vary, then `Constantine` can safely linearize `oFdF` and deliver data invariance: regardless of the contents of the secret `pw`, considering fixed and publicly known array sizes, the trace of data addresses will be the same. If, however, we allow the size of `pw` to vary, then any access past the size of `pw` must be replaced with an access to a valid address, and thus data invariance cannot be guaranteed. Notice that, under the assumption of public sizes, and provided that we can correctly infer the sizes of the objects, our approach delivers the same guarantees as `Constantine`.

Nonetheless, there is a class of programs for which data invariance can always be delivered: the *publicly-safe* programs, which Definition 2.11 shall formalize. Definition 2.11 relies on the notions of *data* and *control* dependency. These concepts are present in Ferrante et al. [1987]'s seminal work. Definition 2.9 revisits them, to keep this paper self consistent.

Definition 2.9 (Data & Control Dependency [Ferrante et al. 1987]). Variable y is *data dependent* on variable x if y is assigned in an instruction that uses x. Variable x is *control dependent* on a variable p if the value assigned to x depends on the outcome of a branch whose condition uses p. Variable x is *dependent* on variable y if either x is data or control dependent on y, or if x is dependent on some variable z, which is either data or control dependent on y.

Example 2.10. Variable `r` in Figure 2 (c) is data dependent on variables `t`, `g`, `pw` and `i`, due to the assignment in Line 8. This variable is control dependent on the predicate `i < n`, which controls the loop at Line 6 of Figure 2 (c).

To delimit the set of codes that we transform, Definition 2.11 establishes two classes of programs: those that are shadow safe and those that are publicly safe. The latter is a proper subset of the former. Notice that in Definition 2.11, shadow safety refers to the expression e used to index a memory access, e.g., $m[e]$. Public safety, in turn, refers to the site, within the program, where the instruction that refers to $m[e]$ exists.

Definition 2.11 (Safety). A program meets *shadow safety* if, for every memory access $m[i]$, *the index i* is not dependent on **secret**. If, in addition, for every *access* $m[i]$ that is control dependent on **secret**, $i < \texttt{size}(m)$ holds, then this program meets *public safety* [Cauligi et al. 2019].

Public safety was initially proposed by Cauligi et al. [2019] to characterize programs whose accesses to data can always be safely linearized. Definition 2.11 provides a weaker version of public safety, which we call *shadow safety*. This notion is important to this paper because, as we shall demonstrate in Section 4.7 (Theorem 4.26), programs that meet shadow safety, once transformed, become "almost" data invariant: data addresses that can vary with secret inputs are replaced with the shadow memory whenever these accesses should not happen and cannot be proven safe. Example 2.12 clarifies the difference between public safety and shadow safety.

Example 2.12. Function `oFdF` in Figure 2 (a) is shadow safe: variable `i`, used to index memory accesses at Line 6, is neither data nor control dependent on any secret. However, the accesses at Line 6 are control dependent on the predicate `g[i] != pw[i]`, which relies on the secret input pw. Therefore, this program will be publicly safe if it can be proven that $\texttt{i} < size(\texttt{g}) \wedge \texttt{i} < size(\texttt{pw})$ (the proof technique is immaterial to this discussion). If these two properties hold, then it follows that `g_i = g->data[i]` and `pw_i = pw->data[i]`, for every $0 \leq i < n$ in the linearized code (Figure 3, Lines 20–25), and thus every trace of accesses to the data cache produced by function `memsafe_oTdT` will be the same (i.e. data invariant).

When transforming publicly-safe programs we guarantee operation and data invariance. We also preserve memory safety. Nevertheless, we emphasize that is not our goal to provide proper solutions for data invariance; rather, our focus is to demonstrate that Moll and Hack's partial control-flow linearization can be adapted to make programs operation invariant with regards to secret inputs. Data invariance comes as a bonus for publicly-safe programs.

3 PARTIAL CONTROL-FLOW LINEARIZATION

Partial control-flow linearization is a code generation technique conceived for the single instruction, multiple data (SIMD) execution model. SIMD machines are an economically viable alternative to process the so-called "embarrassingly parallel" workloads. The model characterizes the stream processors in graphics processing units (GPUs) [Garland and Kirk 2010] or the vector units found in modern CPUs [Chen et al. 2021], like Intel x86's SSE and AVX, AMD's 3DNow!, ARM's NEON, Sparc's VIS, PowerPC's AltiVec and MIPS' MSA. In this environment, multiple *processing elements*, or threads, simultaneously execute the same operation on different data. Example 3.1 will make this *modus operandi* more concrete.

Example 3.1. Figure 4 (a) shows a simplified CUDA kernel — a program meant to run on a graphics processing unit. This program counts how many occurrences of keys stored in the array `q` are present in the matrix `d`. Results are stored in the array `r`. Syntactically, function `search` looks like standard C code. Semantically, it is very different: the program will be executed by multiple threads in lockstep. Although threads see the same arguments `q`, `d` and `r`, they differ with respect to the special register `tid`. This identifier is unique per thread. A common pattern in this environment is to use this register to set up the work that each processing element will carry out. In this example, each thread uses its own `tid` as the index of the value in `q` that must be searched in the matrix `d`. The thread identifier also indicates the position in `r` where each thread will store its answer.

The SIMD model suits very well *straight-line code*, that is, code without branches, because the execution flow of the different processing elements never diverges in this setting. However, programs do have branches over which threads might disagree. In face of divergences, threads still move in lockstep at the hardware level; however, some processing elements stop doing useful work. Typically, predicated instructions are used to ensure that processing elements only write back their results when they run along paths actually taken within the program. Example 3.2 illustrates this trend.

Example 3.2. Figure 4 (b) shows the control-flow graph (CFG) of the kernel seen in Figure 4 (a). This CFG contains two conditional branches at the end of blocks 1 and 2. The former can be determined to be uniform, meaning that threads always take the same decision when executing it; the latter is divergent. There exist standard compiler analyses to separate uniform and divergent branches [Coutinho et al. 2011; Sampaio et al. 2014]. Figure 4 (c) shows a linearization of the CFG in Figure 4 (b) that removes the divergent branch while preserving the uniform one. The store in block 4 still happens, but is *silent*, unless the predicate `p1`, which controls the divergent branch in Figure 4 (b), is true. A silent store writes to memory the same value that was already there. A conditional selector (`ctsel`), guarded by `p1`, determines if the store is silent or not.

```
__global__ void search(int *q, int d[T][N], int *r) {
    for (int i = 0; i < N; i++) {
        if (q[tid] == d[tid][i])
            r[tid] += 1
    }
}
```

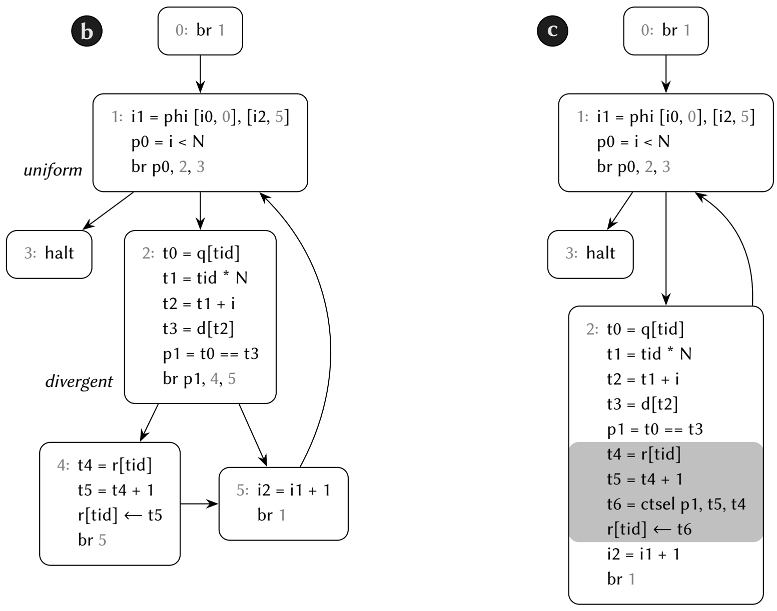


Fig. 4. (a) CUDA kernel that counts occurrences of keys in a matrix. (b) Control-flow graph of the kernel. (c) Partially linearized control-flow graph.

Because only some, but not all, branches in Example 3.2 are removed, the linearization is said to be *partial.* The current state-of-the-art algorithm for partial control-flow linearization is due to Moll and Hack [2018]. Figure 5 shows a version of that algorithm in Python syntax. We present the algorithm for the sake of completeness, for it is extensively described in its original exposition [Moll and Hack 2018]. The algorithm visits the basic blocks in the target graph in a special order: the *compact topological ordering*, which is formalized in Definition 3.3.

Definition 3.3 (Compact Topological Ordering). Given a directed graph G with a unique root vertex s, we say that vertex u *dominates* vertex v if every path from s to v must go through u. An n-sequence of vertices $v_1, \ldots, v_n$ is *dominance compact* if whenever v_1 dominates v_n then v_1 dominates every $v_i, 1 < i < n$. Similarly, an n-sequence of vertices $v_1, \ldots, v_n$ is *loop compact* if whenever v_1 and v_n belong to a loop L then every $v_i, 1 < i < n$, belong to L as well. A topological ordering of the graph G is *compact* if it is both dominance and loop compact with respect to all dominance sets and loops.

Function `compact_order`, in Figure 5 produces a compact topological ordering “`Index`” of the vertices in graph `G`. This function uses an auxiliary routine, `topological_sort`, to produce some topological ordering of the nodes in a graph. Our code also assumes the existence of an “`idom`” relation, such that `idom`(v, u) is true if v is the *immediate dominator* of u. We say that v is the

```
 0 def compact_order(G, entry):
 1     tsort = topological_sort(G)
 2     end = len(tsort)
 3     def schedule(u, start):
 4         bidx = [u]
 5         for i in range(start, end):
 6             v = tsort[i]
 7             if G.idom(v, u): # v is idom.
 8                 bidx += schedule(v, i + 1)
 9         return bidx
10     return schedule(tsort[0], 0)
11
12 def min_index(set, idx):
13     return first(i for i in idx if i in set)
14
```

```
15 def linearize(G):
16     Index = compact_order(G, G.entry)
17     GL = Graph(G.num_vertices)
18     D = set() # The set of deferred edges
19     for b in Index:
20         T = {s for (v, s) in D if v == b}
21         if G.is_uniform(b):
22             for s in G.successors(b):
23                 nxt = min_index(T+{s}, Index)
24                 GL.add_edge(b, nxt)
25                 D = D + {(nxt,t) for t in T+{s}\{nxt}}
26         else: # b is divergent or is unconditional
27             S = G.successors(b)
28             if (S):
29                 nxt = min_index(T+S, Index)
30                 GL.add_edge(b, nxt)
31                 D = D + {(nxt, t) for t in T+S\{nxt}}
32         D = D \ {(v, s) for (v, s) in D if v == b}
33     return GL
```

Fig. 5. Partial Control-Flow Linearization. `linearize(G)` produces a graph `GL` that is a linearized version of `G`. A preprocessing step removes the back-edges in `G` before linearization; hence, `linearize` receives an acyclic graph.

immediate dominator of node u if, and only if, v dominates u, and for any other node t that also dominates u, t also dominates v.

Function `linearize` in Figure 5 builds a graph `GL` that is a linearized version of the graph `G`. Once a compact ordering "*Index*" of the vertices in the CFG is built, the function `linearize`, in Figure 5 visits this sequence of nodes in order. The algorithm keeps a set `D` of *deferred* edges, which are edges that point to *attractors*: vertices that will attract the next nodes yet to be visited. Once attractors are connected to the linearized graph, edges pointing to them are removed from `D`. The successors of uniform branches can still change (Lines 23-26); however, the out-degree of those branches remains the same. Divergent branches undergo more extensive changes: they keep only one successor (Lines 29-32). The links that disappear are added to the set of deferred edges; hence, these successors become attractors to be eventually reintegrated into the linearized graph.

Example 3.4. Figure 6 shows the order in which edges are added to the linearized graph. The original edges are mostly kept, except when node 2 is visited (b = 2) in Line 19 of Figure 5. Node 2 contains a divergent branch. Thus, the node `nxt` of the smallest index among the attractors and successors of 2 is chosen to bear the edge that leaves node 2 (Lines 27-31 in Figure 5). The other successors s of node 2 are marked as targets of edges `nxt` $\longrightarrow s$ in the set `D` of deferred edges.

4 FROM PCFL TO SCE

This section shows how we use partial control-flow linearization (PCFL) to perform side-channel elimination (SCE). This exposition will rely on nomenclature typically adopted in compiler textbooks, which we revisit in Section 4.1. Our explanations happen on top of a minimalistic programming language — subject of Section 4.2. The remaining sections of this paper (§4.3 – §4.7) describe the other steps of the proposed code transformation. Figure 7 lists these steps. We omit two of these phases from our paper, because they have been described in previous work. The array-bounds analysis that we use to infer the length of arrays is described in the work of Sperle Campos et al. [2016]. Our transformation updates the interface of functions to receive these inferred lengths. This step is described in the work of Soares and Pereira [2021]. Our presentation is constructed around the example program in Figure 9, p. 14. The goal of this paper is to convert this program into the semantically equivalent program in Figure 22, p. 27.

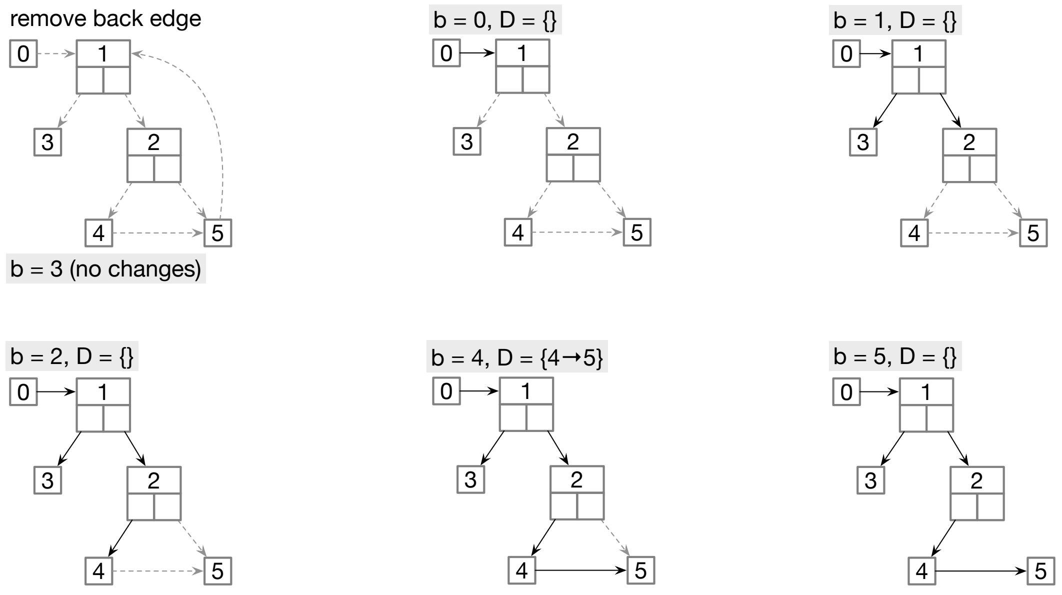


Fig. 6. Sequence of steps that function `linearize` in Figure 5 performs on the program from Figure 4, given that `Index = [0, 1, 3, 2, 4, 5]`.

4.1 Preliminaries

Partial Control-Flow Linearization works on programs containing loops; however, it requires said loops to be *Natural*. Although a natural loop is a well-established concept [Appel 1997], Definition 4.1 revisits it for the sake of completeness. Throughout the paper, we shall adopt the same terminology used in the LLVM documentation[4] when referring to *Natural Loops*.

Definition 4.1 (Natural Loop). A control-flow Graph (CFG) is a directed graph with an entry node *start*. If G is a CFG, then a loop $L \subseteq G$ is a strongly connected subgraph of G. L is called *natural* if it contains a *header* h such that every path from *start* to any node $v \in L$ goes through h.

As a consequence of Definition 4.1, the header dominates all nodes in the loop. Most loops in programs will be natural: they are produced by statements like `while`, `do-while`, `for` and `foreach`. The creation of non-natural loops usually requires abusing the go-to statement. The original description of partial control-flow linearization also requires loops to have unique *latches* (see Definition. 4.2). This requirement can be met for any program via a standard compiler transformation, which we shall not explain further in this paper.

Definition 4.2 (Loop Terminology). A *Forward Edge* is an edge from a node outside the loop to the loop header. A *Back Edge* is an edge from a node inside the loop to the loop header. A *Latch* is the source of a back edge. An *Exiting Edge* is an edge from inside the loop to a node outside of it. The source of an exiting edge is called an *Exiting Block*. Similarly, the destination of an exiting edge is called an *Exit Block*.

4.2 Baseline Language

Figure 8 shows the syntax of the toy language that will be used to explain our ideas. In Figure 8, {} indicates zero or more occurrences, [] denotes optional terms, *id* represents names of variables, *n*

[4]https://llvm.org/docs/LoopTerminology.html

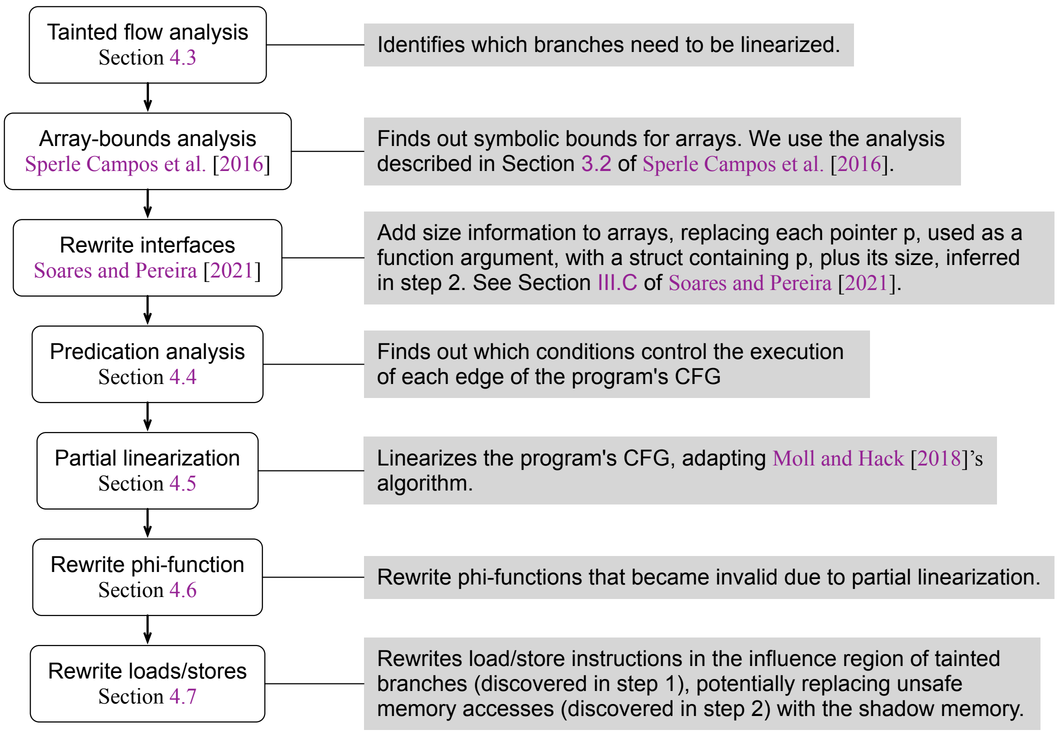

Tainted flow analysis
Section 4.3
Identifies which branches need to be linearized.
Array-bounds analysis
Sperle Campos et al. [2016]
Finds out symbolic bounds for arrays. We use the analysis described in Section 3.2 of Sperle Campos et al. [2016].
Rewrite interfaces
Soares and Pereira [2021]
Add size information to arrays, replacing each pointer p, used as a function argument, with a struct containing p, plus its size, inferred in step 2. See Section III.C of Soares and Pereira [2021].
Predication analysis
Section 4.4
Finds out which conditions control the execution of each edge of the program's CFG
Partial linearization
Section 4.5
Linearizes the program's CFG, adapting Moll and Hack [2018]'s algorithm.
Rewrite phi-function
Section 4.6
Rewrite phi-functions that became invalid due to partial linearization.
Rewrite loads/stores
Section 4.7
Rewrites load/store instructions in the influence region of tainted branches (discovered in step 1), potentially replacing unsafe memory accesses (discovered in step 2) with the shadow memory.

Fig. 7. Overview of the linearization approach proposed in this paper.

stands for numerals, and ℓ ranges over basic block labels. In this paper, we assume all programs to be in the *Static Single Assignment* (SSA) form [Cytron et al. 1989].[5] Thus, every variable has a single definition site and the definition of a variable dominates all its uses. To meet these properties, the toy language is equipped with **phi** functions — special instructions that join multiple definitions of the same variable. In addition, the language provides a **ctsel** (constant-time selector) operation, which is parameterized by a condition c, such that **ctsel** c, v_t, $v_f \equiv v_t$ if $c \equiv$ **true** or v_f otherwise.[6] We represent stores with a left arrow instead of an equal sign to distinguish them from simple assignments. For convenience, we write stores of the form `v[0]` $\leftarrow e$ as `v` $\leftarrow e$. Example 4.3 shows a program in our toy language.

Example 4.3. The code in Figure 9 (a) compares a user's guess g with the secret password pw.[7] It returns true if g equals pw. Function `oFdF` immediately returns false if the test at Line 5 evaluates to true. Hence, `oFdF` might leak the secret password due to the non-constant behavior of the loop at Lines 4–5. Figure 9 (b) shows the implementation of function `oFdF` in our toy language.

4.3 Tainted-Flow Analysis

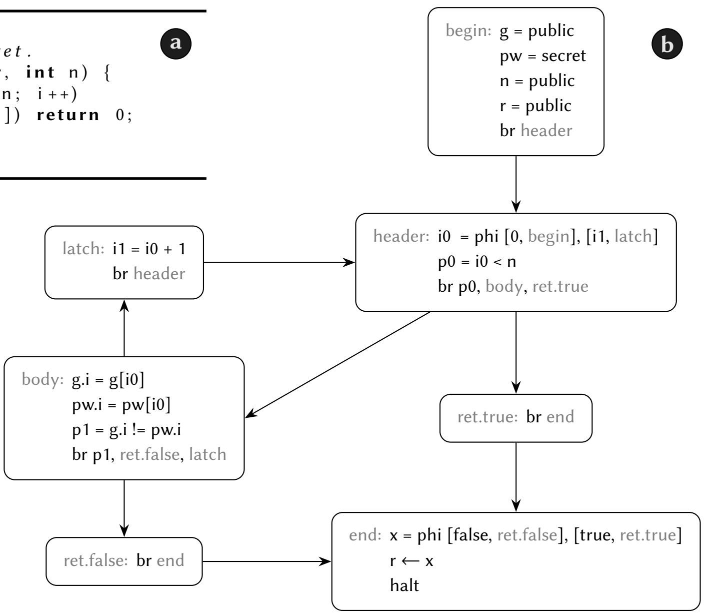
PCFL was first devised to eliminate *divergent* branches from programs. In the context of side-channel resistance, however, we are interested in *tainted* branches. Our toy language defines two

[5]The SSA assumption is not a necessary condition to enable the code transformation described in this paper. Nevertheless, we assume it for convenience, because this format is adopted in the LLVM program representation.
[6]We treat zero as false and any integer different from zero as true.
[7]The code depicted in Fig. 9 is merely used as an example. Bear in mind that passwords should never be stored as plain text.

$$\begin{array}{rl}
\text{Program} & ::= \{\,\text{BasicBlock}\,\} \\
\text{BasicBlock} & ::= \ell\text{: } \{\,\text{Instruction}\,\}\ \text{Terminator} \\
\text{Instruction} & ::= \mathit{id} = \textbf{public} \mid \mathit{id} = \textbf{secret} \mid \mathit{id} = \mathit{Expr} \\
& \mid \mathit{id} = \mathit{id}\,'['\,\text{Value}\,']' \mid \mathit{id}\,'['\,\text{Value}\,']' \leftarrow \mathit{Expr} \\
& \mid \mathit{id} = \textbf{phi}\,'['\,\text{Expr}, \ell\,']'\,\{\,,\ '['\,\text{Expr}, \ell\,']'\,\} \\
& \mid \mathit{id} = \textbf{ctsel}\ \text{Value}, \text{Value}, \text{Value} \\
\text{Terminator} & ::= \textbf{br}\ [\,\text{Value}, \ell,\ ]\ \ell \mid \textbf{halt} \\
\text{Expr} & ::= \text{Value} \mid \mathit{unop}\,\text{Value} \mid \text{Value}\ \mathit{binop}\ \text{Value} \\
\text{Value} & ::= \textbf{true} \mid \textbf{false} \mid n \mid \mathit{id}
\end{array}$$

Fig. 8. Syntax of the baseline language used to design the isochronous transformation. We use prime markers, e.g., ′]′ to delimit terminal symbols.

```
// g (guess) is public.
// pw (password) is secret.
int oFdF(int *g, int *pw, int n) {
    for (int i = 0; i < n; i++)
        if (g[i] != pw[i]) return 0;
    return 1;
}
```



```
begin:    g = public
          pw = secret
          n = public
          r = public
          br header

header:   i0 = phi [0, begin], [i1, latch]
          p0 = i0 < n
          br p0, body, ret.true

latch:    i1 = i0 + 1
          br header

body:     g.i = g[i0]
          pw.i = pw[i0]
          p1 = g.i != pw.i
          br p1, ret.false, latch

ret.true:  br end

ret.false: br end

end:      x = phi [false, ret.false], [true, ret.true]
          r ← x
          halt
```

Fig. 9. (a) Password comparison function that leaks information due to the conditional branch on secret input pw. This code is the same seen in Figure 2 (a) — we copy it to facilitate the understanding of the control-flow graph on the right. (b) Control-flow graph of the password comparison function.

instructions, **secret** and **public**, which we shall use to separate tainted from non-tainted variables. Definition 4.4 categorizes these concepts.

Definition 4.4 (Tainted Information). The *backward slice* of a variable x is the transitive closure of its control and data dependencies (see Definition 2.9). A variable is *tainted* if its backward slice contains a secret value. A branch whose condition uses a tainted predicate is said to be *tainted*. A basic block that ends with a tainted branch is a *tainted block*. A loop that contains an exiting tainted block is a *tainted loop*.

There are standard static analyses that can be used to identify tainted branches [Almeida et al. 2016; Rodrigues et al. 2016]. However, these techniques are orthogonal to the transformation presented in this paper, and shall not be discussed further. In practice, we use the information analysis of Rodrigues et al. to label variables as either tainted or non-tainted. Example 4.5 illustrates how this analysis works.

Example 4.5. Input `pw` is defined as secret in Figure 9. Since the definition of predicate `p1` relies — indirectly, through `pw.i` — on `pw`, the conditional branch at the end of the basic block body is tainted. If a conditional branch is tainted, then the program might contain a timing leak, following the tainted-flow may-analysis of Rodrigues et al.. In the opposite direction, if the program contains no tainted conditional branch, then it does not contain timing leaks, according to that static analysis. One could think that, because the predicate `p1` controls whether or not the execution of the program flows back to the loop header, then the definition of `i0` — which is used to index arrays `g` and `pw` — is control dependent (Definition 2.9) on a secret, and therefore should be tainted (Definition 4.4). This, however, is not the case: the variable `i0` starts with zero and, as the loop executes, is always incremented by one; hence, the value of `i0` *inside* the loop does *not* depend on the branch governed by `p1`. However, if `i0` was used after the loop, then its value outside the loop would be control dependent on `p1`, since the value of `i0` *outside* the loop might vary due to the tainted branch controlled by `p1`. In this scenario, `i0` would be clean inside the loop and tainted after the loop. There is an analogue when we do divergence analysis too: a variable can be uniform inside a loop and divergent outside it. The original formulation of divergence analysis [Sampaio et al. 2014] would use special instructions, called η functions, to split the live ranges of variables inside and outside loops. These different versions of the same variable could then be mapped to a uniform abstract state within the loop and a divergent abstract state outside it. In our case, phi-functions fill the role of η instructions.

4.4 Predication

We let the *side effects* of a program be the set of state modifications that said program carries out on the machine that it controls. In the context of this work, side effects are memory writes. If P is a program and P_l is the partial linearization of P, then we want both P and P_l to have the same side effects when given the same public inputs. To achieve this property, we need to ensure that instructions of P_l only cause side effects when their counterparts in P do. For that, we resort to *predication*, a classic compiler transformation already mentioned in Example 3.2.

In this paper, predication is implemented via the ctsel instruction, whose syntax Figure 8 shows. This instruction is controlled by a boolean condition. To find the boolean associated with each instruction that requires predication, we introduce the notions of *edge* and *block* conditions. We start by formalizing these concepts in Definition 4.6 for *loop-free* programs. Later, in Figure 12 (Page 17), we shall define these conditions for programs with loops.

Definition 4.6 (Block & Edge Conditions). The block condition $BC(v)$ determines when block v executes. The edge condition $EC(u \rightarrow v)$ determines when the edge $u \rightarrow v$ is traversed. The equations in Figure 10 define these mutual relations.

Example 4.7. Figure 11 shows the CFG of a simplified version of function `oFdF` seen in Figure 9 (a), with $\mathsf{n} = 2$ and the loop unrolled. Each edge is labeled with its corresponding edge condition. Block `if.0` always executes; hence, its block condition is `true`. Block `if.1` executes whenever `p0` is false; thus, its block condition is $\overline{\mathtt{p0}}$. The block condition of `end` is the disjunction $(\mathtt{p0} \vee (\overline{\mathtt{p0}} \wedge \mathtt{p1})) \vee (\overline{\mathtt{p0}} \wedge \overline{\mathtt{p1}})$, which reduces to `true`, since block `end` always executes.

$$\frac{terminator(\ell) = \mathsf{br}\, p, \ell', \_}{EC(\ell \longrightarrow \ell') = BC(\ell) \wedge p} \qquad \frac{terminator(\ell) = \mathsf{br}\, p, \_, \ell'}{EC(\ell \longrightarrow \ell') = BC(\ell) \wedge \overline{p}}$$

$$\frac{terminator(\ell) = \mathsf{br}\ \ell'}{EC(\ell \longrightarrow \ell') = BC(\ell)} \qquad \frac{predecessors(\ell) = \{\ell_1, \ldots, \ell_n\}}{BC(\ell) = \bigvee_{j=1}^{n} EC(\ell_j \longrightarrow \ell)}$$

Fig. 10. Block and edge conditions. Function *predecessors*(ℓ) gives the blocks that are predecessors of block ℓ. Function *terminator*(ℓ) returns the last instruction of the block labeled by ℓ (see Fig. 8).

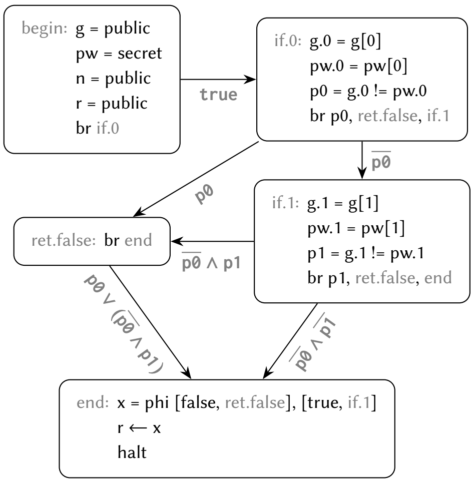


Fig. 11. CFG of function oFdF from Figure 9 (a), with n = 2 and the loop unrolled. Edges are labeled with edge conditions.

4.4.1 From Loop-Free to General Programs. Figure 10 models edge and block conditions of loop-free programs. Once loops are introduced, it becomes necessary to account for the fact that the same control-flow edge might be traversed multiple times. This fact requires adjustments when modeling the conditions associated with loop headers and tainted loop exits. These adjustments will ensure that Property 4.8 holds.

PROPERTY 4.8 (LOOP TERMINATION). *If P_l is the partial linearization of a program P, then any loop of P_l can only terminate due to non-tainted (i.e. public) predicates. If, during the execution of program P, a loop L would have exited through a tainted edge $u \longrightarrow v$, then the rest of the iterations of L in the linearized program P_l shall produce no side effects.*

To model the fact that blocks and edges in loops might be predicated with multiple conditions, we associate these block and edge conditions with a number i. $BC_i(\ell)$ corresponds to the block condition of ℓ at its i-th execution (similarly for EC_i). In this sense, the definitions seen in Figure 10 correspond to EC_i and BC_i, $i \geq 1$. Figure 12 shows the definition of EC_i and BC_i for, respectively, tainted exiting edges and loop headers.

Example 4.9. Figure 13 shows the edge and block conditions, as computed by the rules in Figure 12 when applied over function oFdF, from Figure 9. In this example, we assume that the public parameter n contains the value 2. Thus, the loop iterates twice and the header block is visited three times.

$$\frac{type(\ell \to \ell') = \texttt{Exiting Edge} \quad tainted(p) \quad terminator(\ell) = \texttt{br}\,p, \ell', \_}{EC_{i=1}(\ell \to \ell') = BC(\ell) \wedge p}$$

$$\frac{type(\ell \to \ell') = \texttt{Exiting Edge} \quad tainted(p) \quad terminator(\ell) = \texttt{br}\,p, \ell', \_}{EC_{i>1}(\ell \to \ell') = EC_{i-1}(\ell \to \ell') \vee (BC(\ell) \wedge p)}$$

$$\frac{type(\ell \to \ell') = \texttt{Exiting Edge} \quad tainted(p) \quad terminator(\ell) = \texttt{br}\,p, \_, \ell'}{EC_{i=1}(\ell \to \ell') = BC(\ell) \wedge \overline{p}}$$

$$\frac{type(\ell \to \ell') = \texttt{Exiting Edge} \quad tainted(p) \quad terminator(\ell) = \texttt{br}\,p, \_, \ell'}{EC_{i>1}(\ell \to \ell') = EC_{i-1}(\ell \to \ell') \vee \left(BC(\ell) \wedge \overline{p}\right)}$$

$$\frac{type(\ell) = \texttt{Header} \qquad predecessors_f(\ell) = \{\ell_1, \ldots, \ell_n\}}{BC_{i=1}(\ell) = \bigvee_{j=1}^{n} EC(\ell_j \to \ell)}$$

$$\frac{type(\ell) = \texttt{Header} \qquad predecessors_b(\ell) = \{\ell_1, \ldots, \ell_n\}}{BC_{i>1}(\ell) = \bigvee_{j=1}^{n} EC(\ell_j \to \ell)}$$

Fig. 12. Recursive definitions of block and edge conditions for headers and tainted exiting edges. Function $predecessors_f$ is related to forward edges and $predecessors_b$ to back edges. When written without subscripts, $BC(\ell)$ (similarly for EC) refers to the last execution of block ℓ.

Loop Headers. Figure 10 does not work for loop headers. That is because, when dealing with loops, forward and back edges (see Definition 4.2) play different roles: only forward edges can be traversed to enter a loop and only back edges can be traversed to continue to the next iteration of the loop. But, the original definition of block condition is the *conjunction* of *all* the conditions from incoming edges (see Figure 10). Figure 12 adjusts the definition of block conditions, to distinguish between forward edges and back edges in the case of loop headers as follows: if ℓ_h is a loop header, then $BC_i(\ell_h)$, in iteration i, is the *disjunction* of all the forward-edge conditions (if $i = 1$) or back-edge conditions (if $i > 1$) that reach ℓ_h. Since back-edge conditions implicitly carry the previous block condition $BC_{i-1}(\ell_h)$ of ℓ_h, as soon as one iteration of the loop becomes false, all the subsequent iterations will be false too. Example 4.10 provides more details.

Example 4.10. The edge condition of `begin` $\to$ `header` is true in Figure 13. Hence, if we use Figure 10 to compute BC(`header`), it will always be true. As a result, body would always be able to produce side effects, even when not supposed to. Thus, when computing BC_i(`header`), $i > 1$, the rules in Figure 12 uses `latch` $\to$ `header`, the back edge, but not `begin` $\to$ `body`, the forward edge. The condition of the second execution of `header` becomes $\texttt{p0}_1 \wedge \overline{\texttt{p1}_1}$, and the condition of the third visit becomes $\texttt{p0}_1 \wedge \overline{\texttt{p1}_1} \wedge \texttt{p0}_2 \wedge \overline{\texttt{p1}_2}$. Notice how conditions of previous iterations contribute to compose the condition of the current iteration of the loop.

Tainted Exiting Edges. Figure 10 does not work for tainted exiting edges. Due to Property 4.8, if loop L exits through a tainted edge $u \to v$, then node v should be the only exit with a true block condition in P_l, once L terminates. To meet this property, Figure 12 modifies Figure 10 as follows: if $\ell \to \ell_x$ is a tainted exiting edge, then $EC(\ell \to \ell_x)$ is the *disjunction* of all the edge conditions of $\ell \to \ell_x$ at every iteration of the loop. Thus, as soon as one such condition becomes true for the first time, it will remain true during the entire execution of the partially linearized loop. And, more importantly, no other different tainted edge condition of that loop will ever become true. The predicate that has enabled $\ell \to \ell_x$ disables $\ell \to \ell_L$, the edge that points to inside the loop.

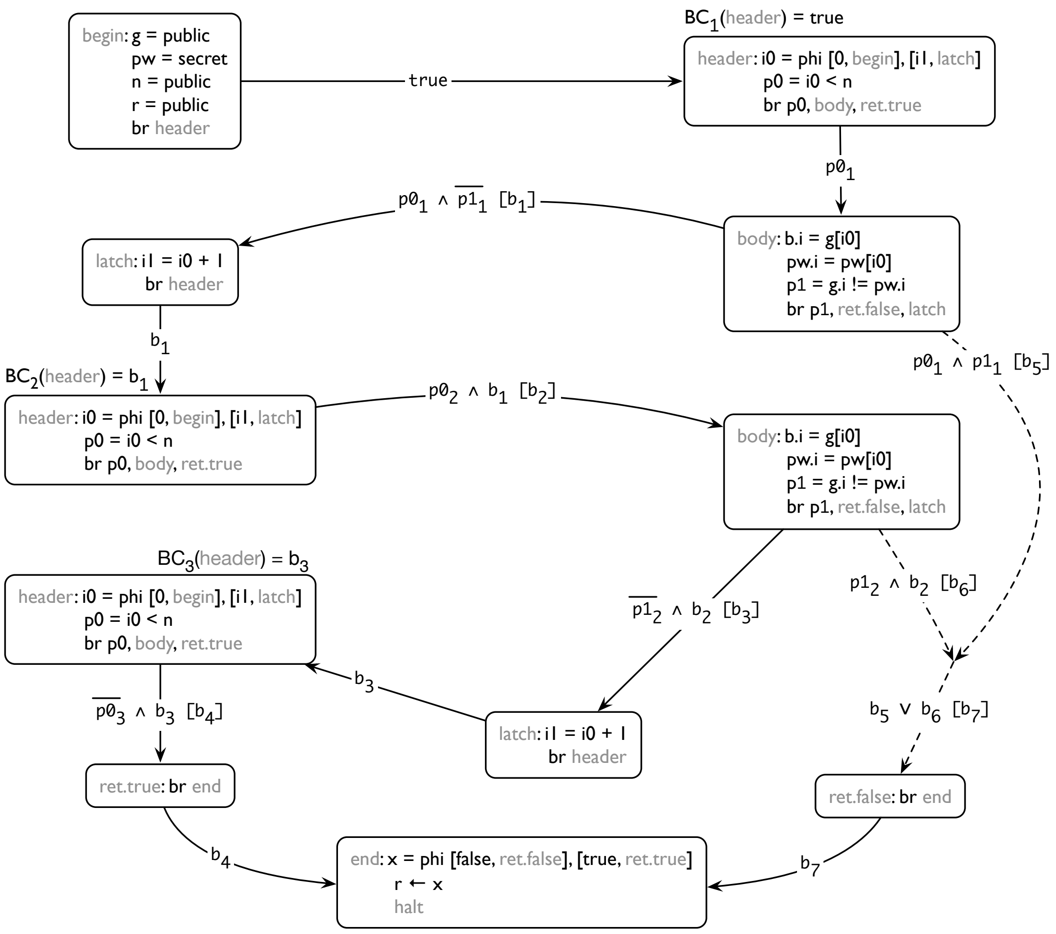


Fig. 13. Block and edge conditions computed on function oFdF from Fig. 9. We assume that n, the public parameters that controls the number of loop iterations, holds value two. We use $e[b]$ on some edges to denote that b is an alias for the boolean condition e, to simplify the figure.

Consequently, the edge condition of $\ell \rightarrow \ell_L$ becomes false, and it is propagated inside the loop, making the next iterations "dummy".

Example 4.11. Going back to Figure 13, lets us assume that EC_1(body → `ret.false`) is true. In other words, the loop should stop thru the tainted exiting edge right in the first iteration. However, Property 4.8 forbids this early exit. Indeed, the code that we shall produce will traverse all the solid edges in Figure 13, but always with false conditions. Thus, from the moment that body is first visited onwards, all the edge and block conditions in the loop become false, while every EC_i(body → `ret.false`), $i > 1$ remains true. As a consequence, EC(`ret.true` → end) is false, for it is the conjunction of three false predicates, and EC(`ret.false` → end) is true, for it includes b_5, which was the first exit condition to be true.

Intuitively, conjunctions are applied whenever a whole set of conditions must hold for an instruction to be executed. For instructions outside loops, we conjunct the set of conditions of every branch leading to that instruction. For instructions inside loops, this set also includes the conditions of different iterations of these loops. In Figure 14, the variables `bc.header`, `bc.body` and `bc.latch` accumulate these conditions. In contrast, disjunctions are used every time the execution of an instruction requires one of the conditions in the set to be true (i.e. when there are multiple paths that lead to that instruction). In the case of tainted edges that exit loops, there are multiple routes that lead to that exit, hence the use of disjunctions. In Figure 14, variable `ec.body_ret.false` accumulates these disjunctions. Notice that disjunctions are not necessary when dealing with clean exiting edges, since that exit will still exist in the final, repaired program.

4.4.2 Materializing Block/Edge Conditions. The rules from Figures 10 and 12 establish how block and edge conditions shall be computed, according to the types of the blocks and edges. However, these conditions cannot be determined statically, and thus we must materialize them into code that will be embedded into the final, repaired program. Example 4.12 illustrates this process.

Example 4.12. Figure 14 shows the program from Figure 9 augmented with code to compute all of its block and edge conditions. Following the rules for loop headers (Figure 12), the block condition of `header` is either the edge condition from `begin` to `header` (forward edge, first iteration), which is always true, or the edge condition from `latch` to `header` (back edge, subsequent iterations), which is just the block condition of `latch`. The edge from `header` to `ret.true` is *clean*, and thus its condition is defined according to the rules from Figure 10: it is the conjunction of the predicate `p0` and the block condition of `header`. In contrast, the edge from `body` to `ret.false` is *tainted*; hence, its condition follows the rules from Figure 12: it is the disjunction of the values of the edge condition from `body` to `ret.false` at each iteration of the loop.

The code from Figure 14 contains extra booleans that keep track of edge and block conditions across iterations of the loop. However, this code is not yet linearized. In the next sections, we show how that program can be partially linearized, and how the booleans mentioned in Example 4.12 are used to predicate the other instructions. As already explained, predication will ensure that original and transformed codes have the same set of side effects.

4.5 Control-Flow Linearization

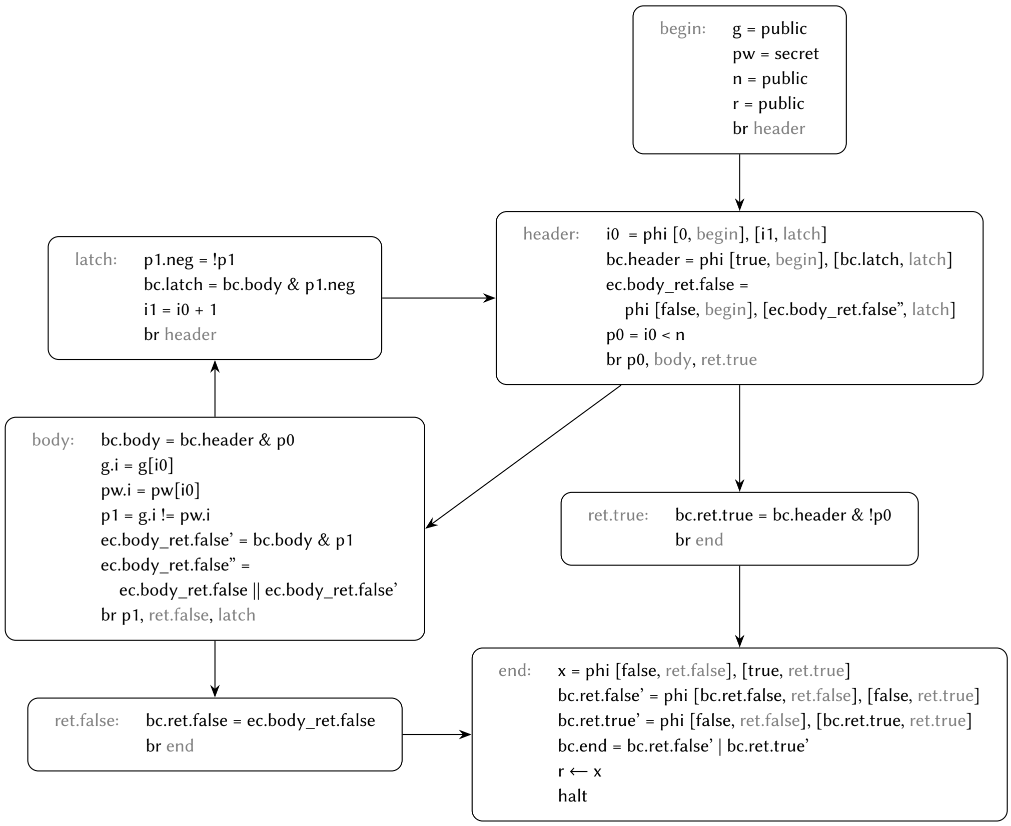
The control-flow graph of the partially linearized program must be rewritten, so that original and transformed programs carry out the same set of side effects. These rewriting rules entail a number of properties concerning the elimination of time-based side channels, which Theorems 4.16 and 4.26 summarize. Additionally, these transformations preserve semantics of terminating programs, as Theorem 4.13 states. Proofs of these theorems are available as supplementary material.

THEOREM 4.13 (SEMANTICS). *Let $T(P) = P'$ be the partial linearization of program P with instructions rewritten, as described in §§4.6 and 4.7. If P' terminates, then programs P and P' produce the same set of side effects.*

Proofs of Lemmas and Theorems are given in Appendix A.

4.5.1 Finding a Compact Ordering. Partial Control-Flow Linearization demands a compact ordering (see Definition 3.3), which implies *dominance compactness* and *loop compactness*. The former is guaranteed by function `compact_order` seen in Figure 5. To attain the latter, we collapse all loops into single nodes, producing a new graph structure with two types of nodes — basic blocks and loops. Loop nodes, by construction, are compact. Hence, we can apply `compact_order` to the root CFG as well as to every loop node and then join the compact orderings obtained, as Example 4.14 shows.

Example 4.14. Figure 15 shows the linearization of the CFG of function `oFdF` from Figure 9. We first collapse the loop formed by nodes `header`, `body` and `latch` into a single node `L`. Loop `L` is



```
begin:      g = public
            pw = secret
            n = public
            r = public
            br header

header:     i0  = phi [0, begin], [i1, latch]
            bc.header = phi [true, begin], [bc.latch, latch]
            ec.body_ret.false =
               phi [false, begin], [ec.body_ret.false”, latch]
            p0 = i0 < n
            br p0, body, ret.true

latch:      p1.neg = !p1
            bc.latch = bc.body & p1.neg
            i1 = i0 + 1
            br header

body:       bc.body = bc.header & p0
            g.i = g[i0]
            pw.i = pw[i0]
            p1 = g.i != pw.i
            ec.body_ret.false’ = bc.body & p1
            ec.body_ret.false” =
               ec.body_ret.false || ec.body_ret.false’
            br p1, ret.false, latch

ret.true:   bc.ret.true = bc.header & !p0
            br end

ret.false:  bc.ret.false = ec.body_ret.false
            br end

end:        x = phi [false, ret.false], [true, ret.true]
            bc.ret.false’ = phi [bc.ret.false, ret.false], [false, ret.true]
            bc.ret.true’ = phi [false, ret.false], [bc.ret.true, ret.true]
            bc.end = bc.ret.false’ | bc.ret.true’
            r ⟵ x
            halt
```

Fig. 14. Function oFdF from Figure 9, embedded with code that compute all of its block and edge conditions, following the rules from Figures 10 and 12.

tainted because one of its exiting blocks — body — is tainted. We then produce a compact ordering of the CFG with its loop collapsed, which Figure 15 (b) shows, as well as a compact ordering of L, shown in Figure 15 (c). Since there is no tainted branch in the loop L, the loop is left unchanged. The compact ordering for the CFG can be seen as the merge of the two compact orderings from (b) and (c): `begin`, `header`, `body`, `latch`, `ret.true`, `ret.false`, `end`.

4.5.2 Rewriting Loop Exits. To deal with divergent loops (i.e. loops with divergent exiting blocks), Moll and Hack [2018] merge every loop exit into a single exiting block at the end of the loop, which becomes the new loop latch. The transformed loop then terminates only when all threads do. This approach, however, leads to dummy execution of instructions that could have been avoided had the public exits been preserved. In this paper, we follow a different path: we redirect every *public* exiting edge of a tainted loop to the first exit block that appears in the compact ordering of the basic blocks. The following example further clarifies our partial linearization:

Example 4.15. Figure 15 (d) shows the linearization of the CFG with the loop collapsed. Notice that there is now a single edge leaving the loop L. This edge corresponds to every public exiting edge of a loop. In this example, there is only one such edge from `header` to `ret.true`, but there could be more. Figure 15 (e) shows the final version of the CFG.

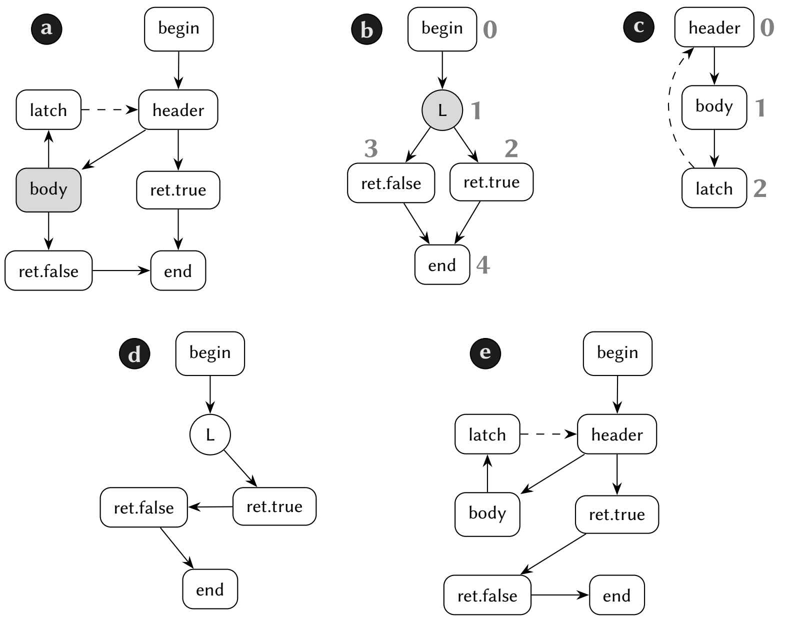


Fig. 15. (a) CFG of oFdF from Figure 9; dashed arrows represent back edges; gray nodes are tainted. (b) The CFG with collapsed loop; numbers indicate the compact ordering. (c) Compact ordering of loop. (d) The collapsed CFG after linearization. (e) Whole CFG after linearization.

In Example 4.15, the tainted branch at node body does not exist after linearization. Thus, the transformed CFG is operation invariant according to Definition 2.2: regardless of secret inputs, it runs the same sequence of instructions. This observation is not exclusive to Example 4.15. As Theorem 4.16 states, partial linearization ensures operation invariance with regard to secret inputs.

Theorem 4.16 (PCFL Gives Operation Invariance). *Let P_l be the partial linearization of P. P_l is operation invariant.*

Partial control-flow linearization is optimal, in the sense that it only modifies tainted branches. In other words, edges departing from branches that are controlled exclusively by public information are not modified. This result follows as a corollary of Theorem C.1 in Moll and Hack [2018], as we state it as Corollary 4.17. In fact, Moll and Hack's result goes a bit beyond Corollary 4.17: there are a few edges departing from tainted branches that shall remain present in the linearized control-flow graph.

Corollary 4.17 (Optimality). *Let ℓ be a basic block within a control-flow graph $G = (E, V)$, such that terminator(ℓ) = $\texttt{br}\,p, \ell_1, \ell_2$ and $\texttt{not}(tainted(p))$ is true. The edges $\ell \longrightarrow \ell_1$ and $\ell \longrightarrow \ell_2$ remain in E after partial control-flow linearization.*

4.5.3 On the Termination of Linearized Programs. As a consequence of Property 4.8, if every exit condition of a loop is controlled exclusively by secret inputs, then linearization will transform it in non-terminating code. If such is the case, then the generation of a non-terminating loop is statically known: this event happens once every branch condition is identified as tainted by the analysis of §4.3. Thus, generation of non-terminating code can be reported back to users. Example 4.18 illustrates this phenomenon.

Example 4.18. The program in Figure 16 (a) contains a loop that has only one exit. This single exit is dependent on secret information, which will be tagged as “tainted" by the flow analysis of Section 4.3. This exit will be removed after linearization. Thus, partial control-flow linearization will produce a program without exiting edges. Figures 17 (a) and (b) show, respectively, the CFG of the loop depicted in function `f` before and after partial control-flow linearization. Notice that, by looking at the CFG from Figure 17 (b), it is possible to detect that the loop will not terminate.

a

```
int f(int secret) {
    ...
    for (int i = 0; i < secret; i++) {
        ...
    }
    ...
}
```

b

```
int g(int public, int secret) {
    ...
    for (int i = 0; i < secret; i++) {
        ...
        if (public) break;
    }
    ...
}
```

Fig. 16. (a) Loop controlled exclusively by secret inputs. (b) Loop controlled by public and secret inputs. This loop will not terminate after linearization if `public == false`.

The general consequence of Property 4.8 is that, after partial linearization, loops can only contain exiting edges dependent on public data. Notice that if the conditions controlled by public data never become true during the execution of the linearized loop, said loop will not terminate. Example 4.19 discusses this possibility.

Example 4.19. The program in Figure 16 (b) contains a loop with two exits: one controlled by tainted information and another controlled by public data. The exiting edge controlled by secret data will be removed by PCFL. Hence, if `public == false`, the linearized loop will run forever.

Discussion. By preventing control flow from leaving loops through edges controlled by tainted data, our implementation of partial control-flow linearization ensures operation invariance with regards to secret values. Previous linearization techniques either forgo loops altogether, as `Lif` [Soares and Pereira 2021] and `SC-Eliminator` [Wu et al. 2018a] do, or handle them dynamically, like `Constantine` [Borrello et al. 2021] does. The approach adopted by `Constantine` avoids non-termination; however, it still allows leakage of secret information. Quoting Borrello et al. [2021]: “*we validate the prediction of the oracle: whenever real program paths wish to iterate more than k times, we adaptively update k allowing the loop to continue, and use the k′ seen on loop exit as the new bound when the program reaches the loop again. The handling of this comparison is also linearized.*" Nevertheless, whenever the control flow reaches the loop current upper bound, information about secret data will leak. In this paper’s approach, the behavior of the loop controlled exclusively by

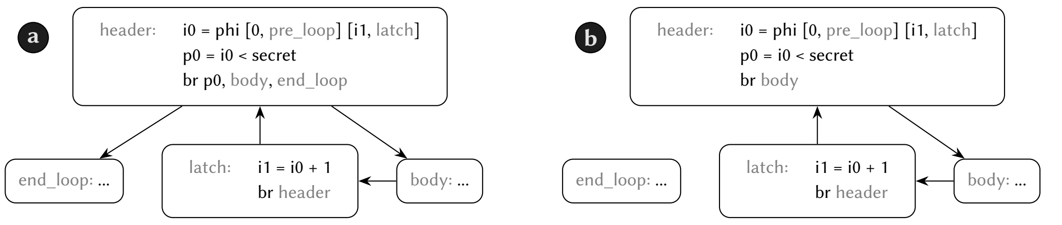


Fig. 17. (a) CFG of the loop from Figure 16 (a) before PCFL. (b) CFG of the loop from Figure 16 (a) after PCFL.

secret data will be the same regardless of these values: the loop will not terminate. Yet, a non-terminating program has problems of its own, at which point the merits of our solution becomes more a question of taste than effectiveness.

4.6 Rewriting Phi-Functions

As we have explained in §4.2, our transformation has been designed to operate on programs in the Static Single Assignment (SSA) Form. The SSA representation uses phi functions to join definitions of variable names that, in the original — pre-SSA transformation — code, would represent the same memory location. The partial linearization of a program P may lead to invalid phi functions in the linearized version P_l, for a predecessor of a block v in P may not be a predecessor of v in P_l. Example 4.20 illustrates this issue.

Example 4.20. Figure 18 (a) shows a CFG before linearization, with block b marked as tainted. Figure 18 (b) shows that CFG after PCFL. Linearization removes the edge $d \longrightarrow g$; hence, reducing the arity of the phi function at g from three to two arguments. The rest of this section will explain how the phi function must be rewritten.

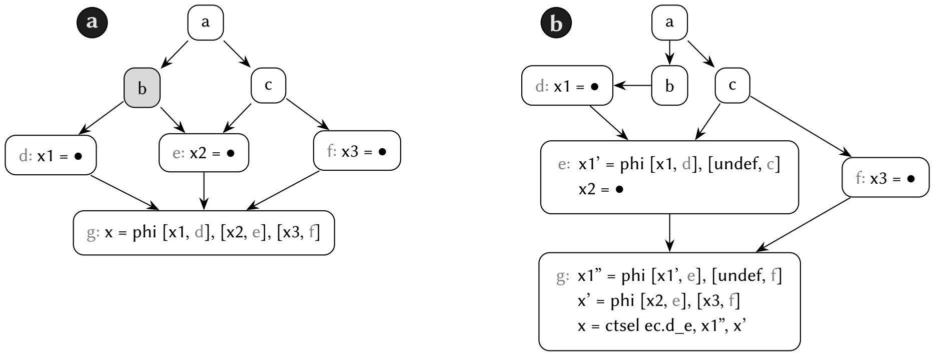

a
a
b
c
d: x1 = ●
e: x2 = ●
f: x3 = ●
g: x = phi [x1, d], [x2, e], [x3, f]
b
a
d: x1 = ●
b
c
e: x1' = phi [x1, d], [undef, c]
x2 = ●
f: x3 = ●
g: x1" = phi [x1', e], [undef, f]
x' = phi [x2, e], [x3, f]
x = ctsel ec.d_e, x1", x'

Fig. 18. (a) CFG before linearization; block b is tainted. (b) CFG after linearization, with the phi function at block g rewritten; variable ec.d_e stores the edge condition of edge $d \longrightarrow g$ from the original graph (see Definition 4.6, Figures 10 and 12).

Phi functions are parameterized by pairs of arguments. Thus, $x = \mathbf{phi}[e_1, \ell_1], [e_2, \ell_2], \ldots, [e_n, \ell_n]$ exists at the beginning of a basic block ℓ, which has n predecessors. If the program flow reaches ℓ coming from ℓ_i, then the phi function indicates that the assignment $x = e_i$ must occur. The difficulty arises because the linearized CFG G_l might not contain the edge that, in the original graph G, would connect ℓ_i to ℓ. To deal with the elimination of edges, we split the arguments of the original phi function into two sets K and R. The former represents the arguments whose corresponding blocks still are predecessors of ℓ in G_l. These arguments are safe to keep without modifications. R contains arguments from blocks that are no longer predecessors of ℓ. The rules that rewrite phi nodes rely on three helper functions *split*, *fill* and *fold*, which Figure 19 defines. Function *Split*, in Fig. 19 separates the arguments of a phi node into the K and R sets. Let φ be the phi function to be transformed and φ_l the new phi function that shall replace φ in P_l. Arguments in K are safe to be transported to φ_l without any modifications, since the predecessor relation has not changed; thus, we fill as much as possible of the arguments of φ_l with K.

Dealing with missing edges. Function *fill* is responsible for moving the old arguments from φ to φ_l (unstarred version, third rule). Once all pairs from K were consumed, we start filling the

$$\frac{K = \{[e_i, \ell_i] \mid \ell_i \rightarrow \ell \in E(G_l)\} \quad R = \{[e_i, \ell_i] \mid \ell_i \rightarrow \ell \notin E(G_l)\}}{split(\mathsf{x} = \mathbf{phi}\,[e_1, \ell_1],\ \ldots,\ [e_n, \ell_n]\ @\ \ell, G_l) = (K, R)}$$

$$\frac{\nexists\,[e_i, \ell_i] \in K\colon \ell_i \in (\ell' \cup \ell s) \quad \exists\,[e_j, \ell_j] \in R\colon reach_G(\ell_j, \ell')}{fill(\ell' \cup \ell s, K, R, G) = [SSAfy(x_j), \ell']}$$

$$\frac{\nexists\,[e_i, \ell_i] \in K\colon \ell_i \in (\ell' \cup \ell s) \quad \nexists\,[e_j, \ell_j] \in R\colon reach_G(\ell_j, \ell')}{fill(\ell' \cup \ell s, K, R, G) = [\mathbf{undef}, \ell']}$$

$$\frac{\exists\,[e_i, \ell_i] \in K\colon \ell_i \in \ell s}{fill(\ell s, K, R, G) = ([e_i, \ell_i], K, R)} \qquad \frac{}{fill(\emptyset, K, R, G) \overset{*}{=} (\emptyset, K \cup R)}$$

$$\frac{fill(\ell s, K, R, G) = ([e_i, \ell_i], K', R') \quad fill(\ell s \setminus \ell_i, K', R', G) \overset{*}{=} (A, U)}{fill(\ell s \neq \emptyset, K, R, G) \overset{*}{=} ([e_i, \ell_i] \cup A, U)}$$

$$\frac{\mathsf{y}_i = \mathbf{ctsel}\, EC(\ell_i \longrightarrow \ell), SSAfy(e_i), x}{fold(x@\ell, [e_i, \ell_i] \cup U) \overset{*}{=} fold(\mathsf{y}_i@\ell, U)} \qquad \frac{}{fold(x@\ell, \emptyset) \overset{*}{=} x}$$

$$\frac{\begin{array}{l} split(\mathsf{x} = \mathbf{phi} \ldots @\ \ell, G_l) = (K, R) \\ fill(predecessors_{G_l}(\ell), K, R, G) \overset{*}{=} (\{[e_1, \ell_1], \ldots, [e_k, \ell_k]\}, U) \\ fold(\mathsf{x} = \mathbf{phi}\,[e_1, \ell_1], \ldots, [e_k, \ell_k]\ @\ \ell, U) \overset{*}{=} \mathsf{z} \end{array}}{rewrite_\phi(\mathsf{x} = \mathbf{phi} \ldots @\ \ell, G, G_l) = \mathsf{z}}$$

Fig. 19. Transformation rule for phi nodes. *inst* @ ℓ indicates that the instruction belongs to the block labeled by ℓ. $E(G)$ are the edges of graph G. *Fold* relies on edge conditions (see Definition 4.6, Figures 10 and 12).

arguments of φ_l with set R. However, the basic blocks in R are not predecessors of ℓ in P_l anymore. Hence, we need to adjust the pairs from R before attaching them as arguments of φ_l. We proceed as follows: if there is a block ℓ_j in R that reaches ℓ' (a predecessor of ℓ in G_l) in the graph G (unstarred version, first rule) then we replace ℓ_j — the original predecessor associated with value x_j — with ℓ'. However, x_j might not be always available in ℓ', which could potentially break the SSA constraints. Hence, we must guarantee that x_j is defined when the program flows to ℓ'; we encapsulated that as function *SSAfy*. If there is no such a block ℓ_j (unstarred version, second rule), we associate the (new) predecessor ℓ' with a special value **undef**, meaning that there is no incoming value for φ that relates to block ℓ'. The starred version of *fill* applies *fill* to all the predecessors of ℓ in G_l and returns the arguments from φ that have not yet been linked to φ_l.

The third and final step for rewriting a phi function is to link with φ_l those arguments from φ that are still unlinked. This is accomplished with function *fold*, which uses the block conditions (see Def. 4.6) of the old predecessors of ℓ to conditionally select between values. The transformation of phi functions is thus the composition of *split*, *fill* and *fold*, and it is represented by function $rewrite_\phi$ in Fig. 19. Function $rewrite_\phi$ returns the variable that shall hold the value of φ in P_l; it is worth noting, however, that there are intermediate instructions that must be added to the basic block as well. Example 4.21 illustrates the transformation of phi nodes.

Example 4.21. The phi function at block g in Figure 18 (a) is rewritten in Fig. 18 (b). It is almost intact, except for the argument [x1, d], because the edge $d \longrightarrow g$ was deleted. A ctsel instruction links the erased argument with the new phi function. The ctsel is parameterized by the edge condition of $d \longrightarrow g$, which in Figure 18 (b) is encoded as variable `ec.d_e`. Two new phi nodes are created to preserve the SSA property, since x1 may not be available in blocks c and f. Notice that x will never be assigned `undef`. To explain why such is the case, let us walk through Figure 18 (b) bottom-up: variable x is either assigned x' or x", out of which only x" can be `undef`. To be assigned `x”`, it is necessary that the edge condition `ec.d_e` is true, which, in turn, requires that edge $d \longrightarrow e$ be taken. Walking upwards, `x”` can only be undef if the edge $f \longrightarrow g$ was taken, and there is no path $d \longrightarrow e \rightsquigarrow f$. Therefore, `x` cannot be assigned `undef` through the incoming value from `f` to `g`. It remains to analyze the case of `x1’`, which can only be assigned `undef` if the edge $c \longrightarrow e$ was taken, and there is no path $d \longrightarrow e \rightsquigarrow c$. In other words, the only path in which `ec.d_e` is true is $a \longrightarrow b \longrightarrow d \longrightarrow e \longrightarrow g$, and it assigns a valid value to `x1`, `x1’`, `x1”` and, finally, `x`.

4.7 Rewriting Memory Operations

The transformation of memory operations has two goals. First, to guarantee that stores only modify state in the linearized program P_l when their counterpart would do the same in the original program P. Second, to prevent the introduction of out-of-bound accesses in P_l. In this work we linearize programs *partially*; hence, not all memory operations need to be modified. To identify which instructions require interventions, we rely on the *influence region* of basic blocks, a notion that Definition 4.22 formalizes. We apply the rules from Figure 20 to loads and stores in the influence region of *tainted* blocks.

Definition 4.22 (Influence Region). Given a directed graph G with a unique exit vertex d, we say that vertex v *post-dominates* vertex u if every path from u to d must go through v. The influence region of a node u is the set of all nodes in paths from u to its post dominator v, excluding u and v.

Example 4.23. The influence region of the tainted block body, from Figure 9 (b), is the set formed by blocks in paths from body to its post dominator `end`: body, `latch`, `header`, `ret.true` and `ret.false`. The block body is within its own influence region because it is inside a loop and thus there are paths from body to end that go through body itself.

$$\frac{\begin{array}{ll}\mathtt{c} = i < size(\mathtt{x}) & \mathtt{c}' = BC(\ell) \mid \mathtt{c} \\ \mathtt{i}' = \textbf{ctsel}\, \mathtt{c}', i, \mathtt{0} & \mathtt{a} = \textbf{ctsel}\, \mathtt{c}', \mathtt{x}, \textbf{shadow}\end{array}}{rewrite_{ld}(\mathtt{y} = \mathtt{x}[i] \ @\ \ell) = \{\mathtt{y} = \mathtt{a}[\mathtt{i}'], \mathtt{a}, \mathtt{i}'\}}$$

$$\frac{rewrite_{ld}(\mathtt{z} = \mathtt{y}[i] \ @\ \ell, G) = \{\mathtt{z}, \mathtt{a}, \mathtt{i}'\} \quad \mathtt{x} = \textbf{ctsel}\, BC(\ell), e, \mathtt{z}}{rewrite_{st}(\mathtt{y}[i] \longleftarrow e \ @\ \ell) = \mathtt{a}[\mathtt{i}'] \longleftarrow \mathtt{x}}$$

Fig. 20. Transformation rules for memory operations. Function $rewrite_{ld}$ preserves loads’ memory safety, while function $rewrite_{st}$ preserves stores’ memory safety and make stores sound with respect to whether they should or not take effect. Both rules rely on block conditions (see Definition 4.6, Figures 10 and 12). *inst* @ ℓ indicates that the instruction belongs to the block labeled by ℓ.

Loads. Function $rewrite_{ld}$, in Figure 20, takes the original load and returns a new load that is memory safe, plus the base address and the index that compose the new access. For that, we follow Soares and Pereira [2021]’s approach: we replace memory accesses that should not occur — i.e. the block condition is false — and are not safe with accesses to a *shadow* address. To determine whether an access is safe or not, we need the size of the structure being manipulated. This can be obtained in multiple ways, e.g. by inferring the size or by asking the user to provide it. Figure 20 abstracts away this computation by relying on a function named *size*. If the size of the value cannot be determined, the access is still guaranteed to be safe, but it becomes data variant (see Theorem 4.26). In practice, we conservatively estimate the size of LLVM arrays without user intervention, using the array-size inference analysis of Sperle Campos et al. [2016].

Example 4.24. The load `pw.i = pw[i0]` in Figure 9 (b) must be transformed, because it is in the influence region of a tainted block (see Example 4.23). Let `bc.body` store the block condition of body and `pw.size` store the size of `pw`. Then, function $\mathit{rewrite}_{ld}$ from Figure 20 transforms the original code in Figure 21 (a) into the code in Figure 21 (b).

a
```
pw.i = pw[i0]
```

b
```
   c = i0 < pw.size
  c' = bc.body | c
  j0 = ctsel c',i0,0
   a = ctsel c',pw,shadow
pw.i = a[j0]
```

c
```
pw[i0] ← x
```

d
```
following load seen  in part (b):
   x' = ctsel bc.body,x,pw.i
a[j0] ← x'
```

Fig. 21. (a) Original load from Fig. 9. (b) Transformed load. (c) Example of a store. (d) Transformed store.

Stores. Function $\mathit{rewrite}_{st}$, in Figure 20, takes the original store and produces a new store that is both memory safe and sound with respect to side effects. We first create a *safe* load to get the current value stored in that memory region (or in the shadow memory, depending on the circumstances). Then, we use the block condition $BC(\ell)$ to select between the current value and the value to be stored: if $BC(\ell)$ is true, the original store is performed, updating the value under that address and producing a side effect; otherwise, the store is silent. Example 4.25 wraps up the transformation of loads and stores presented in this section.

Example 4.25. Suppose that we had a store like `pw[i0] ← x` in block body in Figure 9 (b). Following function $\mathit{rewrite}_{st}$ from Figure 20, we first create a safe load, as shown in Example 4.24. For convenience, let us reuse `pw.i`. The store will then be rewritten from the original code seen in Figure 21 (c) into the sequence in Figure 21 (d).

Like previous work [Borrello et al. 2021; Cauligi et al. 2019; Soares and Pereira 2021], we cannot transform a program that contains memory indexation data dependent on secret inputs. In other words, we cannot transform the following code: `int foo(secret v, int m[]): return m[v]`. Quoting Cauligi et al. [2019], the above code is not "*publicly safe*" (see Definition 2.11) and cannot be made data invariant (see Definition 2.4) using control-flow linearization. Data invariance, as stated by Theorem 4.26, is guaranteed whenever the original program is publicly safe. If the program is shadow safe, then data invariance is not guaranteed, as Example 2.12 (Page 8) has explained.

Theorem 4.26 (The Data Contract). *Let $T(P) = P'$ be the partial linearization of P with loads and stores rewritten after Figure 20. If P is publicly safe, then P' is data invariant. If P is shadow safe, then either P' is data invariant or there exist two input instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$, $\mathcal{S}_1 \neq \mathcal{S}_2$, with corresponding traces of memory addresses τ_1 and τ_2 such that $\tau_1[i] =$ **shadow** or (exclusive) $\tau_2[i] =$ **shadow**, for some i.*

Example 4.27. Figure 22 shows the transformed version of function `oFdF` that we obtain after applying PCFL onto the control-flow graph seen in Figure 9 (b). Variables `g.size` and `pw.size` hold the sizes of the arrays `g` and `pw` whenever function `oFdF` is invoked. If the length of the array is not known statically, then its `size` variable is initialized with zero. Notice that the length does not have to be a constant: it can be a symbolic expression. Every expression used to index an array in the transformed code is compared against the length of that array. If the comparison returns false and the block condition is false, the special variable `shadow` is used as a surrogate address.

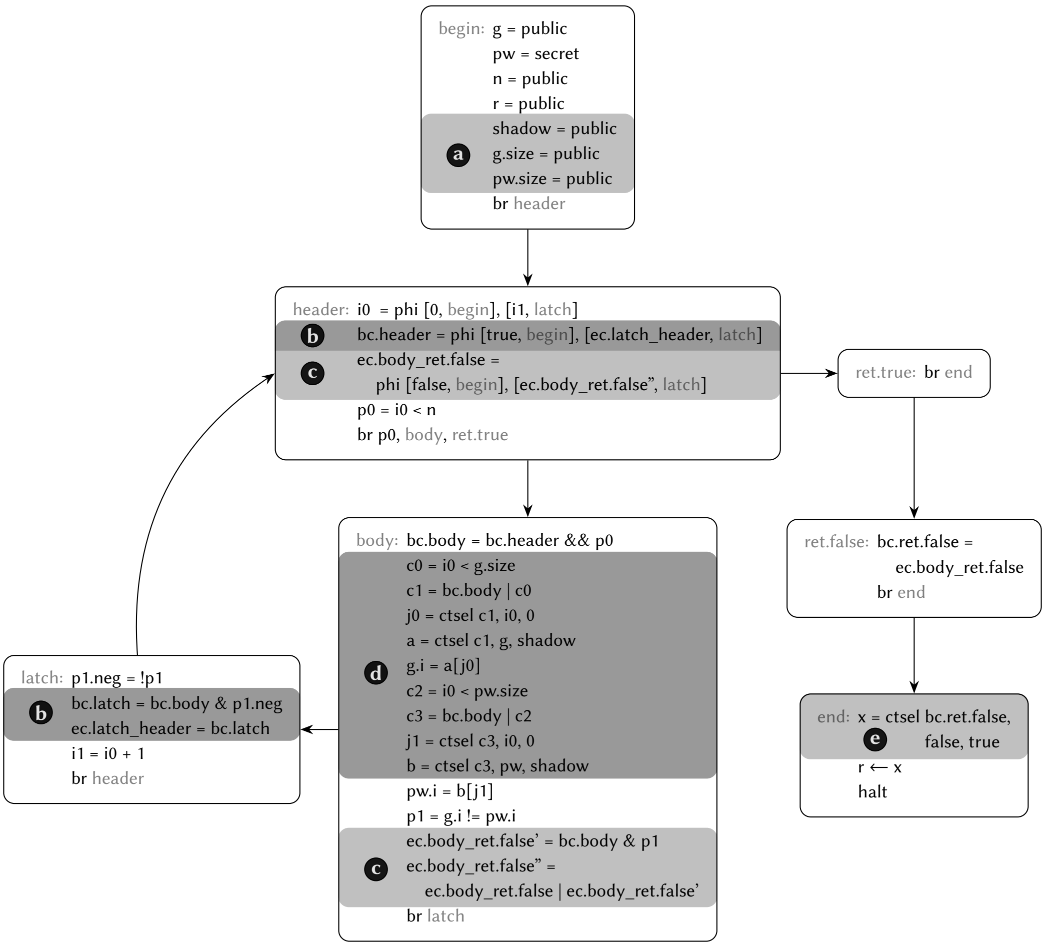

begin: g = public
pw = secret
n = public
r = public
a shadow = public
g.size = public
pw.size = public
br header
header: i0 = phi [0, begin], [i1, latch]
b bc.header = phi [true, begin], [ec.latch_header, latch]
c ec.body_ret.false = phi [false, begin], [ec.body_ret.false”, latch]
p0 = i0 < n
br p0, body, ret.true
ret.true: br end
body: bc.body = bc.header && p0
d c0 = i0 < g.size
c1 = bc.body | c0
j0 = ctsel c1, i0, 0
a = ctsel c1, g, shadow
g.i = a[j0]
c2 = i0 < pw.size
c3 = bc.body | c2
j1 = ctsel c3, i0, 0
b = ctsel c3, pw, shadow
pw.i = b[j1]
p1 = g.i != pw.i
c ec.body_ret.false’ = bc.body & p1
ec.body_ret.false” = ec.body_ret.false | ec.body_ret.false’
br latch
latch: p1.neg = !p1
b bc.latch = bc.body & p1.neg
ec.latch_header = bc.latch
i1 = i0 + 1
br header
ret.false: bc.ret.false = ec.body_ret.false
br end
end: x = ctsel bc.ret.false, false, true
e
r ⟵ x
halt

Fig. 22. Function oFdF from Figure 9 after PCFL and with the instructions rewritten. (a) The shadow memory and the size of input pw, used in the transformation of a tainted load. (b) Computation of the block condition of the loop header (§4.4, Figure 12). (c) Computation of the edge condition of the tainted exiting edge body $\longrightarrow$ `ret.false` (§4.4, Figure 12). (d) Predication of load instructions to ensure memory safety (§4.7, Figure 20). (e) Transformation of a phi function (§4.6).

5 EVALUATION

This section evaluates the techniques described in this paper through five research questions:

RQ1: By how much does the proposed approach increase code size?
RQ2: What is the running time of applying the proposed transformation onto programs?
RQ3: How does the proposed approach impact the running time of programs?
RQ4: What are the security guarantees achieved by the proposed approach?
RQ5: How the general C programs compiled with the proposed approach compare with programs written in a domain-specific language for constant-time cryptography?

To provide perspective on our results, we compare them with those produced by `Lif` [Soares and Pereira 2021], `FaCT` [Cauligi et al. 2019], `SC-Eliminator` [Wu et al. 2018a] and `Constantine` [Borrello et al. 2021]. The last two tools aim at making programs data and operation invariant with regard to their secret inputs. Our PCFL and `Lif`, in turn, only guarantee operation invariance, although when handling publicly safe programs they also ensure data invariance. Code written in the `FaCT` domain-specific language is, by construction, publicly safe; hence, `FaCT` delivers both operation and data invariance for this class of programs. When presenting results for `SC-Eliminator` and `Constantine`, we show how these tools fare with and without data-flow protection.

Hardware. Experiments run on an Intel(R) Core(TM) i5-1035G1 4-Core Processor, clocked at 3.6 GHz. L1 data and instruction caches have 128 KB. Main memory has 8 GB.

Software. The above hardware runs in Linux Manjaro 21.2.5 (5.16.14-1-generic x86_64). Our program transformation plus Soares and Pereira [2021]'s (`Lif`) are implemented in LLVM 13.0. We use a version of `Lif` downloaded from https://github.com/lac-dcc/lif/tree/artifact/cgo. We use a version of `Constantine` downloaded from https://github.com/pietroborrello/constantine, which is implemented in LLVM 9.0. `SC-Eliminator` is available as an ACM artifact at https://zenodo.org/record/1299357 using LLVM 3.9.1. However, due to problems to reuse that artifact, we have downloaded `SC-Eliminator` directly from https://bitbucket.org/mengwu/timingsyn, and have updated it to use LLVM 13.0. We use a version of `FaCT` downloaded from https://github.com/PLSysSec/FaCT, which uses libraries from LLVM 6.0. These tools were downloaded on December 6th, 2021. Thus, results from this paper can be directly compared with `Lif` and `SC-Eliminator`, for both use LLVM 13.0 to obtain the original bytecode that will be transformed. `Constantine`, on the other hand, uses LLVM 9.0; hence, the starting bytecode file is not guaranteed to be the same.

To detect examples of operation variance, we use `CTgrind`, available at https://github.com/agl/ctgrind, and `CFGgrind`, available at https://github.com/rimsa/CFGgrind. These tools are `Valgrind` [Nethercote and Seward 2007] plugins. To detect examples of data variance, we insert instrumentation in the transformed programs to print the addresses that they access during execution. Instrumentation is inserted in the intermediate representation of programs, via a pass that we have implemented. In addition of linearizing the control-flow graph of programs, `Constantine` and `SC-Eliminator` also try to eliminate memory-based side channels. `SC-Eliminator` does it by preloading data in the beginning of functions; `Constantine` does it by traversing the entire buffer whenever a cell within said buffer is read or written. Such interventions make code much slower and much bigger. Thus, for fairness, we show results for these tools with and without data preloading. It is our understanding that, once data linearization is disabled, `SC-Eliminator` and `Constantine` have the same purpose as our implementation of PCFL.

Benchmarks. We use 13 benchmarks, including seven from Wu et al. [2018a][8]. Each benchmark has at least two inputs, with parts tagged as either public or secret. Two of the benchmarks, `ssl3` and `donna`, are from the `FaCT` repository. We have translated them into C. We implemented the four remaining benchmarks: `hash-one`, `plain-many`, `plain-one` and `log-redactor`, to exercise control-flow constructs absent in Wu et al.'s collection. The benchmark `plain-one` corresponds to the example that we have been using throughout the paper (Figure 9). Programs `hash-one` and `plain-many` are variations of `plain-one`.

We handle, without any user intervention, all these benchmarks, except `loki91`. The original implementation of `loki91` contains a loop whose every exit is controlled by sensitive information; hence, it is inherently leaky. To make `loki91` amenable to linearization, we had to modify this

[8]Section 5.5 uses one more program: `crypto-secretbox`, which we translated from `FaCT`. We omit this program from the remaining experiments, because it cannot be linearized by tools other than our implementation of PCFL

loop with a terminating condition based on public information. This condition is the maximum 16-bit integer, i.e., 32,767 iterations. Figure 23 illustrates this change. The transformation is safe: once the secret condition triggers, further iterations become innocuous. In other words, as a consequence of linearization, the loop stops producing side effects. The modification in Figure 23 was implemented manually, but we could have performed it automatically, if necessary to apply it on more benchmarks. However, such pattern — a loop controlled exclusively by secret information — is rare. Our implementation of PCFL handles all the 21 benchmarks used by Soares and Pereira [2021] that are also available in Wu et al. [2018a]'s and Borrello et al. [2021]'s artifacts without modification (again, except for `loki91`).

```
int f(int secret) {
    ...
    for (int i = 0; i < secret; i++) {
        // computation
    }
    ...
}
```

```
const int LIMIT = 32767;
int modified_f(int secret) {
    ...
    for (int i = 0; i < LIMIT; i++) {
        if (i >= secret) continue;
        // computation
    }
    ...
}
```

Fig. 23. (a) Loop controlled exclusively by secret inputs (as shown in Figure 16). (b) Version of the loop modified to ensure termination — similar change was performed in the benchmark `loki91`.

Numbers reported in this section refer to the transformed program after it is optimized with `LLVM opt -O3`. The evaluation of security guarantees (§5.5) uses these optimized programs, in binary format. The benchmarks contain code to initialize inputs and print outputs; however, our numbers refer only to their kernels. `Lif` and `SC-Eliminator` cannot transform three benchmarks due to unbounded loops: `donna`, `ssl3` and `loki91`. One of the other benchmarks, `plain-many`, when transformed by `Constantine`, `Lif` and `SC-Eliminator` crashes at running time. Furthermore, for benchmark `ssl3`, `Constantine` produces code whose output is incorrect.

5.1 RQ1: Size of Transformed Code

Figure 24 reports the size of the programs produced by the different tools that we evaluate in this paper. Size is measured by counting LLVM instructions in the intermediate representation of the transformed kernel after optimizations run.

Considering only the nine benchmarks that `SC-Eliminator` and `Lif` handle, we generate 3,393 LLVM instructions. The original version of `SC-Eliminator` produces 97,772 instructions, whereas `SC-Eliminator`'s CFL gives 76,004. `Lif` generates 138,079 instructions. It is worth remembering that both `SC-Eliminator` and `Lif` were applied to a version of the programs with loops entirely unrolled, for neither of the two tools can handle general loops; hence the higher number of instructions. The complete implementation of `Constantine` (CFL + DFL) yields 4,182 LLVM instructions, while `Constantine`'s CFL outputs 3,293. When compiled with LLVM 13.0 at -O3, the original 13 benchmarks add up to 4,130 instructions, 2,977 of which corresponds to the subset of nine benchmarks handled by all tools. In relative terms, our partial linearization increases code size by 1.33x. Considering only the 11 benchmarks correctly handled by `Constantine`, `Constantine`'s original and CFL-only implementations increase code size by, respectively, 2.73x and 2.64x.

5.2 RQ2: Transformation Time

Figure 25 shows the time (in milliseconds) that each technique evaluated in this paper takes to transform programs. To provide the reader a baseline, we also show the time that LLVM takes to

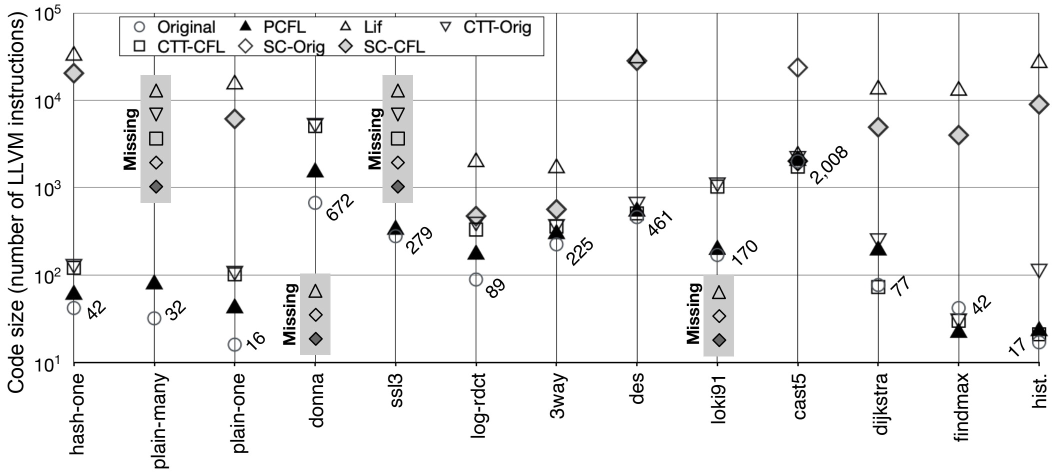


Fig. 24. Code size (in number of LLVM instructions) of transformed programs. Symbols in gray boxes show tools that are missing for particular benchmarks. CTT refers to Constantine, SC refers to SC-Eliminator. Orig refers to these two tools as originally implemented. CFL refers to these two tools with control-flow linearization only — thus, closer to our implementation of PCFL in purpose. The figure uses log scale; hence, for reference purposes, we include the actual size of the original programs, when compiled with LLVM 13.0 at the -O3 optimization level.

apply all the optimizations at the -O3 level onto these programs. Numbers refer only to the time taken by opt, LLVM's optimizer, to run passes onto LLVM intermediate representation: it does not include time to parse C or to generate machine code — roughly the same for all the approaches.

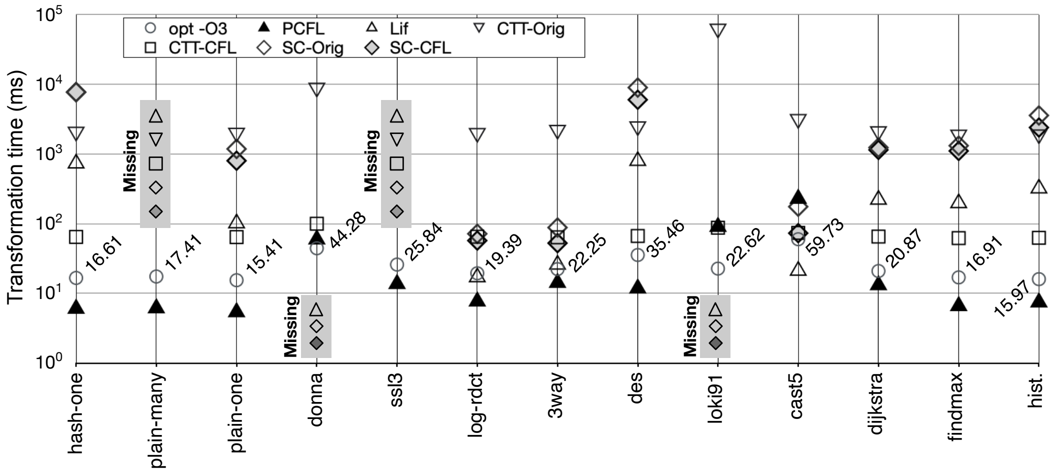


Fig. 25. Time (in milliseconds) to apply each transformation onto the benchmarks. To facilitate comparison, the figure explicitly includes values for the time taken to run LLVM opt -O3 on each benchmark. The gray boxes mark benchmarks that some tools could not handle.

Considering only the nine benchmarks that all tools can deal with, it takes, on average, 24.73 ms to apply opt -O3 onto each benchmark, without any transformation. Our PCFL technique takes about 33.49 ms per benchmark. Lif, not counting the time to unroll loops, takes 271.61 ms. The original implementation of SC-Eliminator takes 2,697.06 ms, whereas SC-Eliminator's CFL takes 2,149.28 ms. SC-Eliminator and Lif are slower because they operate on larger programs, due to unrolling. Constantine's CFL takes about 65.19 ms per benchmark. However, when we consider the entire script used to apply Constantine — which includes everything from profiling up to all the transformations that it applies — this number grows up to 2,045.22 ms. This total is the summation of several independent passes that Constantine applies — some of them coming from LLVM's accompanying tools. Thus, LLVM bytecodes are read and traversed multiple times. We understand that were these passes grouped into a single LLVM pass, Constantine could run faster.

5.3 RQ3: Performance of Transformed Code

Figure 26 shows the running time of transformed programs. Timing the benchmarks was challenging: except for loki91, they all run under less than 1 ms. We executed each benchmark 20 times per tool, removed the two fastest and the two slowest samples per benchmark and averaged the remaining 16 samples. We then used Student's t-test [Gosset 1908] to check for statistically significant results across all the three populations, pair-wisely. Assuming a confidence level of 99%, and considering only the nine benchmarks that SC-Eliminator and Lif can handle, we could observe five results where the programs produced by our implementation of PCFL run faster than those generated by SC-Eliminator's CFL and six results where our approach performs better than both SC-Eliminator's original implementation and Lif. Lowering the confidence interval to 0.95, we observe one additional benchmark for which our approach yields code that runs faster than the code produced by SC-Eliminator's CFL. The mean overhead introduced by both SC-Eliminator and our PCFL is of 1.61x. The CFL-only version of SC-Eliminator adds less overhead onto programs: 1.42x. Lif, on the other hand, generates code that is 3.74x slower. Nonetheless, the expressive overhead promoted by Lif was hugely influenced by the benchmark des (8.32x slower).

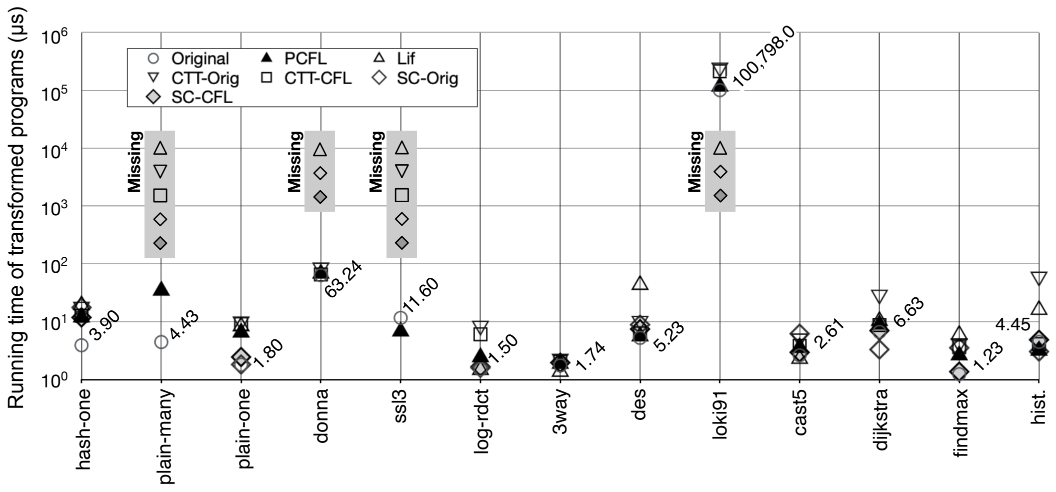


Fig. 26. Running time (in microseconds) of transformed programs. To facilitate comparison, the figure explicitly shows the running time of the original programs when compiled with LLVM 13.0. Symbols in gray boxes show missing tools for particular benchmarks.

Assuming a confidence level of 99%, our technique performs better than both Constantine's original and CFL-only implementations in six out of the 11 benchmarks correctly handled by Constantine. By lowering the confidence interval to 0.95, we could observe one additional benchmark for which we produce faster code than Constantine (both versions). The overhead introduced by Constantine with respect to the nine benchmarks that all tools can correctly deal with was larger than ours: 4.62x for the original implementation and 1.94x for Constantine's CFL-only version. Similarly to the case of Lif, the huge overhead introduced by Constantine's original tool was heavily influenced by the benchmark histogram (12.26x slower). If we consider the 11 benchmarks that Constantine handles, then its average overhead is of 2.24x (original) and 2.12x (CFL), whereas the overhead imposed by our implementation of PCFL is of 1.17x — which is also the mean overhead caused by our tool if we consider all the 13 benchmarks. However, most of the overhead caused by Constantine happens in loki91. This is the benchmark with the largest number of branches: 76, out of which 16 are tainted (See Borrello et al. [2021]'s Table 1). Constantine uses more instructions than our implementation of PCFL: the linearized code that it produces, after optimizations, contains 1,031 LLVM instructions, whereas the code that we produce contains 196. The effects of this difference are amplified by the large trip count of loki91's core loop: 32,767 iterations.

5.4 R4: Security Evaluation

To investigate if the different techniques evaluated in this section are strengthening the security guarantees of programs, we use three tools: CTgrind [Langley 2010], CFGgrind [Rimsa et al. 2021], and an LLVM pass that instruments load and store operations. CTgrind and CFGgrind are Valgrind plugins. The former checks if a program contains a branch that reads data tainted by secret information. The latter counts the number of binary instructions executed per function. The LLVM pass, a tool of our own craft, inserts instrumentation to print the addresses accessed by the load and store operations present in the LLVM intermediate representation of programs. We compare the strings of addresses produced by programs fed with two distinct inputs. Different strings demonstrate data variance. Figures 27 and 28 subsume the results of the security analyses.

CTgrind: Columns Opr and Opr3 of Figure 27 summarize the results of CTgrind's analysis. CTgrind is a dynamic analysis tool; hence, it requires running each program. These programs come with two inputs each. This experiment uses both. CTgrind reports that SC-Eliminator failed to achieve operation invariance for two benchmarks: hash-one and histogram. We analyzed the LLVM intermediate files produced by SC-Eliminator and confirmed that SC-Eliminator indeed failed to linearize some of the tainted branches. In several benchmarks, CTgrind reports that code produced by Constantine is not operation invariant. Debugging the code produced by Constantine is harder than analyzing the code produced by SC-Eliminator, because Constantine's code transformations are more extensive. We inspected the code that Constantine produced for plain-only, our smallest kernel. In that case, Constantine uses memory cells whose purpose is similar to our shadow memory. These blocks of memory are not initialized, but might be used in conditional operations. In this case, CTgrind issues warnings because it considers as tainted any memory that is not initialized. In spite of these results, we emphasize that the absence of warnings issued by CTgrind does not demonstrate that the code has been correctly linearized: it is still possible that different inputs cause tainted branches to be executed.

Data Variance: When probing data variance in Figure 27, we use an LLVM instrumentation pass to verify if the sequence of addresses accessed by each kernel is the same, regardless of the input. In this case, we consider the original versions of Constantine and SC-Eliminator, not the versions that only do control-flow linearization. The column data in Figure 27 reports the

	Original			PCFL				Lif				Constantine				SC-Eliminator			
	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3
hash-one	∞	N	N	Y	SH	Y	Y	Y	0	Y	Y	Y	0	N	N	Y	∞	N	N
plain-many	∞	N	N	Y	SH	Y	Y	X	X	X	X	X	X	X	X	X	X	X	X
plain-one	∞	N	N	Y	SH	Y	Y	Y	SH	Y	Y	Y	∞	N	N	Y	2048_{14}	Y	Y
donna	∞	N	N	Y	0	Y	Y	UL	UL	UL	UL	Y	∞	N	N	UL	UL	UL	UL
ssl3	0	N	N	Y	0	Y	Y	UL	UL	UL	UL	X	X	X	X	UL	UL	UL	UL
log-rdct	∞	N	N	Y	SH	Y	Y	Y	SH	Y	Y	Y	∞	N	N	Y	0	Y	Y
3way	∞	N	N	Y	0	Y	Y	Y	0	Y	Y	Y	0	Y	Y	Y	0	Y	Y
des	∞	N	N	Y	0	Y	Y	Y	0	Y	Y	Y	56_5	Y	Y	Y	0	Y	Y
loki91	N/A	N	N	Y	N/A	Y	Y	UL	UL	UL	UL	Y	N/A	Y	Y	UL	UL	UL	UL
cast5	788_{12}	Y	Y	Y	788_{12}	Y	Y	Y	788_{12}	Y	Y	Y	60_5	Y	Y	Y	60_5	Y	Y
dijkstra	∞	N	N	Y	SH	Y	Y	Y	SH	Y	Y	Y	0	N	N	Y	∞	N	N
findmax	∞	N	N	Y	0	Y	Y	Y	0	Y	Y	Y	0	Y	Y	Y	0	Y	Y
histogram	∞	N	N	Y	396_8	Y	Y	Y	396_8	Y	Y	Y	60_5	N	N	Y	396_8	N	Y

N	**N**o: execution is not operation invariant
Y	**Y**es: execution was operation invariant
X	Tool/transformed program crashes
UL	Program contains **U**nbounded **L**oop and cannot be linearized
NA	**N**ot **A**vailable: the trace is too large to be processed
∞	Data traces contain a different number of addresses
SH	At least one pair of corresponding addresses uses **S**hadow **M**emory

Fig. 27. Security guarantees achieved by the different tools. `Cor` indicates if the transformed program produces the same output as its original counterpart. `Data` is the largest difference between data addresses in corresponding positions in the traces produced by the two inputs without compiler optimizations. The subscript next to each number is the index of the most significant bit where addresses have diverged. `Opr` refers to operation invariance without compiler optimizations, given two different inputs. `Opr3` refers to operation invariance at the `LLVM opt -O3` optimization level, again, considering two different inputs.

maximum difference between corresponding addresses in the traces produced when the programs run with the two different inputs. We could not compute this result for `loki91`: log generation stops once traces reach 26.3GB of size. If the traces contain a different number of addresses, column `data` reports ∞. If all the addresses match up perfectly, column `data` shows the number zero. If code produced by `Lif` or PCFL accesses the shadow memory, column `data` reports "SH". Any difference in these traces necessarily have one of the addresses equal to the shadow memory slot, which is always the same address. Traces cover every address accessed by instructions in the linearized parts of instrumented programs.

Two benchmarks, `cast5` and `histogram`, are not shadow safe. Both use tables indexed by secret inputs. As an example, `histogram` counts the frequency of characters in the sensitive input by incrementing cells within an array of integers, e.g., `c[sensitive_char % SIZE]++`. `Constantine` and `SC-Eliminator` can still generate data-invariant code to programs that are not shadow safe, assuming the hit-miss threat model seen in Section 2. For instance, if we assume[9] that differences of less than 64 in code produced by either `Constantine` or `SC-Eliminator` will be in the same cache line, then benchmarks like `des` and `cast5` should be considered safe after linearization. Five binaries produced via PCFL fail to achieve complete data invariance because some memory accesses are replaced with the shadow memory, which is a unique address for the entire program. Nevertheless, we could verify that the data contract stated in Theorem 4.26 holds for all the 11 benchmarks that are shadow safe. Furthermore, in all 13 benchmarks transformed via PCFL, traces of data

[9]Notice that assuming that address differences smaller than 64 fit in the same cache line is an oversimplification. If we have 64B memory blocks, two addresses are in the same cache line (considering a typical set-associative architecture) whenever they differ only in their last 6 address-bits. Thus, we can have addresses that differ less than 64 which still belong to different memory blocks. As an example, `0b00111110` = 62 and `0b01000001` = 65 are less than 64 apart, but belong to blocks `0b00xxxxxx` and `0b01xxxxxx`. To provide a more informative metric, Figure 27 also reports the maximum among indices of the most significant differing bit between pairs of addresses.

addresses have always the same length — a direct consequence of operation invariance. As a final observation in regards to data invariance, we ported `ssl3` and `donna` from `FaCT`. When written in `FaCT`, every benchmark is publicly safe, and we succeed in delivering the same guarantees as that domain-specific language: complete non-interference between secret inputs and addresses accessed in the instruction and data caches.

`CFGgrind`: Figure 28 reports the number of instructions fetched during the execution of the linearized functions. In this case, we use `CFGgrind` to count only instructions that exist within the scope of linearized functions[10]. This dynamic analysis does not prove operation invariance; however, a divergence in the number of fetched instructions demonstrates the absence of this property. We show results only for non-optimized programs, for the sake of space. Nevertheless, the same qualitative results (column `opr`) remain true for codes optimized with clang -O3.

Important: the number of instructions counted in this table is not a proxy for the speed of linearized code.

	Original		PCFL		Lif		CTT-Orig		CTT-CFL		SC-Eliminator	
	input-1	input-2	input-1	input-2	input-1	input-2	input-1	input-2	input-1	input-2	input-1	input-2
hash-one	12,303	12,309	60,486		53,245		66,579		28,712		6,158	
plain-many	8,208	679	240,960		X	X	X	X	X	X	X	X
plain-one	6,153	3,078	36,914		23,578		95,265	47,649	40,987	20,507	5,084	
donna	26,464	26,784	205,824		UL	UL	127,220		107,220		UL	UL
ssl3	7,856	7,793	26,820		UL	UL	X	X	X	X	UL	UL
log-rdct	613	421	7,830		3,408		22,533	8,784	5,813	4,200	693	
3way	496	531	1,584		2,266		663		421		830	
des	16,580	16,536	56,022		46,676		35,539		18,501		12,233	
loki91	125,893,642	125,893,220	1,153,493,104		UL	UL	8,668,376,240		565,568,000		UL	UL
cast5	262		262		802		28,721		7,160		3,437	
dijkstra	5,227	5,413	43,295		19,510		1,941,609		37,759		3,809	4,022
findmax	5,023	5,025	27,029		17,004		3,786		2,027		3,005	
histogram	39,021	39,024	90,065		102,555		3,245,012		798,024		14,255	

UL Program contains **U**nbounded **L**oops; hence, cannot be linearized   X Tool crashes or transformed program crashes

Fig. 28. Number of instructions fetched during the execution of functions linearized by different tools. `input-1`: number of instructions executed with the first input. `input-2`: number of instructions executed with the second input. Whenever `input` − 1 = `input` − 2, we merge the two columns. Because `SC-Eliminator` outlines part of the linearized code, we can count only a small fraction of the code that it generates. As a consequence, the number of instructions measured for the code transformed by `SC-Eliminator` is the same if we use it with our without data linearization.

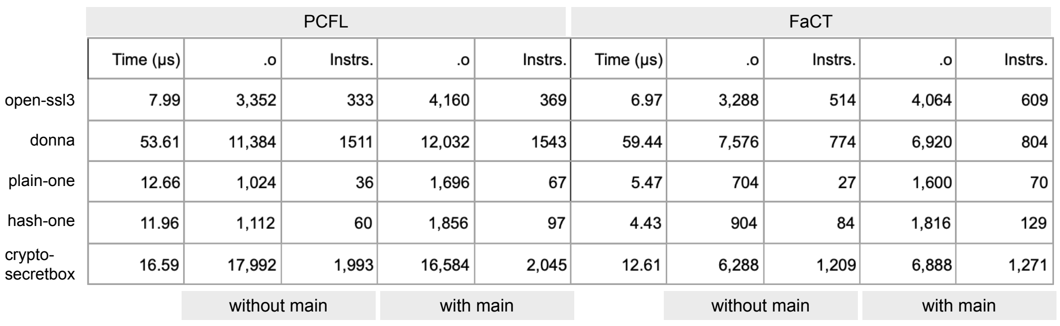
In all the thirteen benchmarks, codes partially linearized using our technique always fetch the same number of instructions. Such is also the case for programs that could be compiled with `Lif`. We failed to observe operation invariance in one program produced by `SC-Eliminator`: `dijkstra`, confirming the warning issued by `CTgrind`. Two programs transformed by `Constantine`, `plain-one` and `log-redact`, did not process the same number of instructions. These issues have been reported back to Borrello et al. According to Pietro Borrello, one of `Constantine`'s authors, the leak happens because the input used to test the tool forces more iterations of the linearized

[10]The numbers that Figure 28 show is not a proxy for running time of linearized programs. In this experiment, we cannot use `CFGgrind` to count the number of instructions fetched during the execution of the entire binary file. It is necessary to probe only linearized functions because the ELF executables contain code that is not touched by linearization. For instance, the code inserted by the `lld` linker (like runtime symbol resolution and the procedure linkage table), or the code invoked from external libraries (mostly to implement input/output). In particular, we experimented difficulty to count instructions that `SC-Eliminator` linearized. It outlines large chunks of the linearized code, and we cannot, at the binary level, distinguish code linearized by `SC-Eliminator` from code that it has not touched. Thus, Figure 28 shows only a small part of the code transformed by `SC-Eliminator`, namely, the instructions that remain in the original linearized functions.

loop than the input used to train it. In Borrello's words: "`Constantine` *is expecting such a case, and dynamically updates the loop-controlling variable if it observes the loop being executed more than it expects, and at the next iteration, the loop will be executed the new maximum number of times. This incurs a one-time side channel but mitigates the fact of Constantine not being fully trained with worst-case conditions*[11]".

5.5 RQ5: Comparison with a Domain-Specific Language

Our implementation of partial-control flow linearization in LLVM in practice gives developers the chance to obtain in C (or other languages that LLVM supports) the same security guarantees provided by `FaCT` [Cauligi et al. 2019]. However, contrary to C, `FaCT` deals with less general control-flow constructs. Currently, `FaCT` supports three control-flow statements: `if-then-else`; `for-from-to` and `return`. Our implementation of PCFL, in contrast, handles the whole of the LLVM intermediate representation, whose unstructured control flow subsumes C/C++'s, including constructs absent in `FaCT`, such as `do-while`, `break`, `continue` and `goto`[12]. This section compares programs written in `FaCT` with similar codes written in C, and linearized with PCFL. This evaluation uses five benchmarks. Three of them, `openssl-ssl3`, `donna` and `crypto-secretbox` were taken from the `FaCT` repository and translated to C. The other two, `plain-one` and `plain-hash`, were translated from C to `FaCT`. Except for `crypto-secretbox`, all these programs are used in the experiments that we report in previous sections. We omit `crypto-secretbox` from those experiments because none of the other tools, `Lif`, `Constantine` or `SC-Eliminator`, can linearize it.



	PCFL					FaCT				
	Time (µs)	.o	Instrs.	.o	Instrs.	Time (µs)	.o	Instrs.	.o	Instrs.
open-ssl3	7.99	3,352	333	4,160	369	6.97	3,288	514	4,064	609
donna	53.61	11,384	1511	12,032	1543	59.44	7,576	774	6,920	804
plain-one	12.66	1,024	36	1,696	67	5.47	704	27	1,600	70
hash-one	11.96	1,112	60	1,856	97	4.43	904	84	1,816	129
crypto-secretbox	16.59	17,992	1,993	16,584	2,045	12.61	6,288	1,209	6,888	1,271
		without main		with main			without main		with main	

Fig. 29. Comparison between programs written in C and linearized with PCFL, and similar programs written in `FaCT`, using equivalent control-flow structures. The column `.o` shows the size, in bytes, of the binary object file. The column `Instrs` shows the number of instructions in the LLVM representation of each program. Because they use different `main` functions, we show results with and without this routine. All these programs, once converted to the LLVM representation, are optimized with `opt -O3`.

The programs written in `FaCT` are, usually, shorter and faster. However, the programs that we generate contain code to ensure memory safety, like the use of the shadow memory, as in Figure 21 (b). Therefore, every linearized load operation in a program produced with our technique will contain four extra instructions absent in the equivalent `FaCT` program. Moreover, the transformation of a store operation requires a load to read the current value in memory. Because this load is also safe, the four-instruction overhead also applies. In `FaCT`, developers use a type qualifier, `assume`(e), to specify that the expression e is the upper bound of an array. This clause works as a contract: the

[11]P. Borrello, personal communication, December 29th, 2022.
[12]PCFL has one restriction concerning the `goto` statement: *loops must be natural.* The implication of this fact is that PCFL will not linearize a loop with multiple entry points. In the C programming language, such loops can be created with `goto`.

compiler does not generate code to ensure in-bounds memory accesses; rather, the programmer promises that buffer limits will be respected. Thus, it is possible to provoke out-of-bounds access in the `FaCT` program, because contracts are not verified. To this effect, we have forced out-of-bounds accesses in `plain-one` — something that cannot happen in the binary produced with PCFL.

Programmability. In our experience, porting C programs to `FaCT` was relatively difficult: only two benchmarks, `plain-one` and `plain-hash`, admitted straightforward translation to `FaCT`. The main challenge is dealing with control-flow constructions that are absent in the DSL. Porting `FaCT` programs to C, in turn, was easier, albeit time consuming. Nevertheless, even when translation is simple, the benchmarks written in C are not strictly equivalent to the programs written in `FaCT`. `FaCT` contains builtin functions that do not translate directly to C:

- The builtin function `view`, which returns a slice of an array. We simulate it with pointer arithmetics: the upper limit of a “view" is given by a base address plus an offset.
- The builtin function `len` that returns the length of an array. To replace it, our C benchmarks use a struct with two fields: a pointer `data` to the array, and an unsigned integer `size`, denoting the size of that array (See Figure 3). In most programs, our static analysis suffices to associate size information with the array itself. When such is not possible, memory accesses in non-executed paths are replaced with accesses to the shadow memory.
- The `declassify` function, which marks data as “public". We could not simulate this feature in C; hence, our code will contain more linearized parts than the code produced from `FaCT` programs that use it.

6 RELATED WORK

This paper draws its contributions from two different communities: high-performance computing and software security. Concerning the former, this work is related to research about control-flow linearization. Concerning the latter, it is related to the static elimination of side channels. In this section, we explain how the paper connects with previous contributions in these two domains.

Partial Control-Flow Linearization. In its essence, Moll and Hack [2018]'s algorithm for control-flow linearization is an efficient way to support predication, inasmuch as it spares uniform branches from being predicated. Predication, as already explained in §3, is a technique to convert control dependencies into data dependencies. To the best of our knowledge, the first description of a systematic way to perform predication is due to Allen et al. [1983], although the problem had already been described in earlier work [Towle 1976; Wolfe 1978]. After Allen et al.'s original foray in the field, predication has been refined and expanded in many different ways, and today is standard textbook material [Clements 2013].

The fact that control-flow linearization was already a concern almost 40 years ago makes it surprising that Moll and Hack's algorithm took so long to emerge. Compared to previous work, PCFL enjoys a number of advantages. First, when compared to Ferrante and Mace [1985]'s well-known linearization approach, Moll and Hack's algorithm has better complexity (linear vs log-linear). Second, it is substantially simpler than previous approaches of similar service, such as Karrenberg and Hack [2012]'s. In the words of Moll and Hack: “*Karrenberg and Hack's method spans over five algorithm listings*", whereas the PCFL routine is fully described by the 33 lines of Python in Figure 5. Finally, PCFL handles unstructured control flows, in contrast to heuristics used in practice [Moreira et al. 2017] by the Intel SPMD Compiler, for instance.

Nevertheless, we emphasize that this paper is not about the design of a partial control-flow linearization approach. We reuse Moll and Hack's algorithm almost without modifications. Our changes in the algorithm have been described in §4.5. There exists only one important difference between Moll and Hack's implementation and ours, which is a consequence of the different purpose that we have when using partial control-flow linearization. In Moll and Hack's context, loops only terminate when all threads exit it; thus, the linearized loop contains only one exit block, at its end. Moll and Hack add phi functions to identify through which exit each thread left the loop. This approach is similar to what `Constantine` does: it computes an upper bound for the loop and forces execution up to this trip count. In our case, a loop can have multiple exits: indeed, any exit that is only dependent on public data will be left untouched by our transformation.

Side Channel Elimination via Control-Flow Linearization. Timing attacks attracted much attention during the nineties [Dhem et al. 1998; Kocher 1996; Kocher et al. 1999; Wray 1992]. However, the problem was known before [Singel 1976]. The literature contains many examples of protections against such attacks. This paper is concerned with the so-called *white-box* mitigations, which require intervening in the software. Wu et al. [2018a] calls such methodologies *program repair*. For an overview of *black-box* approaches, such as defenses implemented at the operating-system level, we refer the reader to the comprehensive discussion presented by Cock et al. [2014].

The current state-of-the-art techniques that make programs operation invariant via some form of control-flow linearization are Soares and Pereira [2021]'s and Borrello et al. [2021]'s works. The former is fully static; the latter combines static and dynamic analyses. As mentioned in §1, our approach is more general than Soares and Pereira's, because it handles programs with loops. When dealing with loop-free programs where all branches are tainted, these techniques are equivalent. If the loop-free program contains non-tainted branches, then our technique will not linearize them, whereas Soares and Pereira's will. In terms of operation invariance, we expect our method to handle the same programs that `Constantine` does; however, `Constantine` also protects the data cache, something that we only do for publicly safe programs. Nevertheless, whereas `Constantine` requires executing the program, our approach is fully static.

7 CONCLUSION

The key contribution of this work is to adapt a vectorization technique — partial control-flow linearization — to solve an open question in side-channel resistance: the static elimination of operation-based side channels in general programs while preserving branches controlled by public inputs. We believe that the techniques discussed in this paper let a programmer write, directly in C, code that meets the same safety properties of algorithms written in the `FaCT` [Cauligi et al. 2019] domain-specific language. As we have discussed in §5, our side-channel elimination technique performs favorably when compared with `Lif` [Soares and Pereira 2021] and `Constantine` [Borrello et al. 2021], state-of-the-art tools released in 2021. Yet, in contrast to `Lif`, it handles programs with unbounded loops, and in contrast to `Constantine`, it delivers operation invariance statically.

Software. Our prototype is available at https://github.com/lac-dcc/lif.

REFERENCES

Johan Agat. 2000. Transforming out Timing Leaks. In *POPL*. Association for Computing Machinery, New York, NY, USA, 40–53. https://doi.org/10.1145/325694.325702

J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *POPL*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *SEC*. USENIX Association, USA, 53–70. https://doi.org/10.5555/3241094.3241100

J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *Security & Privacy*. IEEE, New York, NY, USA, 965–982. https://doi.org/10.1109/SP40000.2020.00028

Andrew W. Appel. 1997. *Modern Compiler Implementation in Java*. Cambridge University Press, USA.

Musard Balliu, Mads Dam, and Roberto Guanciale. 2014. Automating Information Flow Analysis of Low Level Code. In *CCS*. Association for Computing Machinery, New York, NY, USA, 1080–1091. https://doi.org/10.1145/2660267.2660322

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.* 4, POPL, Article 7 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371075

Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *CCS*. Association for Computing Machinery, New York, NY, USA, 462–476. https://doi.org/10.1145/3460120.3484761

Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS* (Virtual Event, Republic of Korea). Association for Computing Machinery, New York, NY, USA, 715–733. https://doi.org/10.1145/3460120.3484583

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 174–189. https://doi.org/10.1145/3314221.3314605

S. Chattopadhyay and A. Roychoudhury. 2018. Symbolic Verification of Cache Side-Channel Freedom. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2812–2823. https://doi.org/10.1109/TCAD.2018.2858402

Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *ASPLOS* (Virtual, USA). Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. 2012. Compiler Mitigations for Time Attacks on Modern X86 Processors. *Trans. Archit. Code Optim.* 8, 4, Article 23 (Jan. 2012), 20 pages. https://doi.org/10.1145/2086696.2086702

Alan Clements. 2013. *Computer Organization and Architecture: Themes and Variations*. Cengage Learning, USA.

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Timing Channels on SeL4. In *CCS*. Association for Computing Machinery, New York, NY, USA, 570–581. https://doi.org/10.1145/2660267.2660294

Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. 2011. Divergence Analysis and Optimizations. In *PACT*. IEEE, USA, 320–329. https://doi.org/10.1109/PACT.2011.63

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *POPL*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. 1998. A Practical Implementation of the Timing Attack. In *CARDIS (Lecture Notes in Computer Science, Vol. 1820)*, Jean-Jacques Quisquater and Bruce Schneier (Eds.). Springer-Verlag, Berlin, Heidelberg, 167–182. https://doi.org/10.1007/10721064_15

Alexander Fell, Hung Thinh Pham, and Siew-Kei Lam. 2019. TAD: Time Side-Channel Attack Defense of Obfuscated Source Code. In *ASP-DAC*. Association for Computing Machinery, New York, NY, USA, 58–63. https://doi.org/10.1145/3287624.3287694

Jeanne Ferrante and Mary Mace. 1985. On Linearizing Parallel Code. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/318593.318636

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *Transactions on Computers* 21, 9 (sep 1972), 948–960. https://doi.org/10.1109/TC.1972.5009071

Michael Garland and David B. Kirk. 2010. Understanding Throughput-Oriented Architectures. *Commun. ACM* 53, 11 (nov 2010), 58–66. https://doi.org/10.1145/1839676.1839694

William Sealy Gosset. 1908. The Probable Error of a Mean. *Biometrika* 6, 1 (March 1908), 1–25. https://doi.org/10.1093/biomet/6.1.1 Originally published under the pseudonym "Student".

Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory. In *SEC*. USENIX Association, USA, 217–233. https://doi.org/10.5555/3241189.3241208

Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *Security & Privacy*. IEEE, New York, US, 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

Ralf Karrenberg and Sebastian Hack. 2012. Improving Performance of OpenCL on CPUs. In *CC*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/978-3-642-28652-0_1

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*. Springer-Verlag, Berlin, Heidelberg, 104–113. https://doi.org/10.1007/3-540-68697-5_9

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO*. Springer-Verlag, Berlin, Heidelberg, 388–397. https://doi.org/10.1007/3-540-48405-1_25

Adam Langley. 2010. CTGrind—checking that functions are constant time with Valgrind. https://github.com/agl/ctgrind

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. IEEE Computer Society, Washington, USA, 75. https://doi.org/10.5555/977395.977673

Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 543–556. https://doi.org/10.1145/3192366.3192413

Rubens E.A. Moreira, Caroline Collange, and Fernando Magno Quintão Pereira. 2017. Function Call Re-Vectorization. In *PPoPP*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/3018743.3018751

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Security and Privacy*. IEEE, Washington, DC, USA, 710–728. https://doi.org/10.1109/SP.2017.53

Willard Rafnsson, Limin Jia, and Lujo Bauer. 2017. Timing-Sensitive Noninterference through Composition. In *POST*. Springer-Verlag, Heidelberg, Germany, 3–25. https://doi.org/10.1007/978-3-662-54455-6_1

Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is My Code Constant Time?. In *DATE*. European Design and Automation Association, Leuven, BEL, 1701–1706. https://doi.org/10.23919/DATE.2017.7927267

Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. 2021. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* 51, 2 (2021), 353–384. https://doi.org/10.1002/spe.2907

Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse Representation of Implicit Flows with Applications to Side-Channel Detection. In *CC*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/2892208.2892230

Diogo Sampaio, Rafael Martins de Souza, Caroline Collange, and Fernando Magno Quintão Pereira. 2014. Divergence Analysis. *Trans. Program. Lang. Syst.* 35, 4, Article 13 (2014), 36 pages. https://doi.org/10.1145/2523815

Ryan Singel. 1976. Declassified NSA Document Reveals the Secret History of TEMPEST about (TEMPEST: A Signal Problem). *Cryptologic Spectrum* 2, 3 (1976), 26–30.

Luigi Soares and Fernando Magno Quintão Pereira. 2021. Memory-Safe Elimination of Side Channels. In *CGO*. IEEE, Washington, USA, 200–210. https://doi.org/10.1109/CGO51591.2021.9370305

Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of Function Arguments. In *CC*. Association for Computing Machinery, New York, NY, USA, 163–173. https://doi.org/10.1145/2892208.2892225

Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. 2019. Quantitative Mitigation of Timing Side Channels. In *CAV*. Springer, Heidelberg, Germany, 140–160. https://doi.org/10.1007/978-3-030-25540-4_8

Ross Albert Towle. 1976. *Control and Data Dependence for Program Transformations.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign, USA. AAI7624191.

J. Van Cleemput, B. De Sutter, and K. De Bosschere. 2020. Adaptive Compiler Strategies for Mitigating Timing Side Channel Attacks. *Transactions on Dependable and Secure Computing* 17, 1 (2020), 35–49. https://doi.org/10.1109/TDSC.2017.2729549

Michael Joseph Wolfe. 1978. *Techniques for improving the inherent parallelism in programs – Master Thesis.* Master's thesis. University of Illinois at Urbana-Chaimpain.

John C. Wray. 1992. An Analysis of Covert Timing Channels. *J. Comput. Secur.* 1, 3–4 (may 1992), 219–232. https://doi.org/10.1109/RISP.1991.130767

Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018a. Eliminating Timing Side-Channel Leaks Using Program Repair. In *ISSTA*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3213846.3213851

Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018b. *[ISSTA '18 Artifact Evaluation] Eliminating Timing Side-Channel Leaks Using Program Repair.* Zenodo. https://doi.org/10.5281/zenodo.1299357

Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *CSFW*. IEEE, USA, 5. https://doi.org/10.5555/872752.873524

Rui Zhang, Michael D. Bond, and Yinqian Zhang. 2022. Cape: Compiler-Aided Program Transformation for HTM-Based Cache Side-Channel Defense. In *CC*. Association for Computing Machinery, New York, NY, USA, 181–193. https://doi.org/10.1145/3497776.3517778

A PROOFS OF THEOREMS

This section contains proofs for the theorems in §4, which we have omitted from the main body of the paper. We shall henceforth refer to properties about partial control-flow linearization from Moll and Hack [2018]. For more details about the proofs, we refer the reader to Moll and Hack's original presentation. The first property that we revisit talks about the correspondence between paths in the original CFG G and in the partially linearized CFG G_l, and it is stated as follows:

Theorem A.1 (Path Embedding [Moll and Hack 2018, Theorem 3.2]). *For each path $\pi \in G$, there is a path $\pi_l \in G_l$ such that π is a subpath of π_l. By subpath, we mean that the nodes seen in π can be found in π_l in the same order that they appear in π.*

Lemma A.2 says that PCFL, as described in this paper, does not add to nor remove any blocks from the CFG of a program. We write $V(G)$ for the vertices of a graph G.

Lemma A.2 (PCFL Preserves Basic Blocks). *Let G_l be the partial linearization of CFG G. Then, $V(G) = V(G_l)$.*

Proof. The proof follows from the PCFL algorithm defined in Fig. 5 and the approach for handling tainted loops described in §4.5.2, which differs from Moll and Hack's method. □

Lemma A.4 states that any block executed in program P has a corresponding block that is active in the linearized program P_l when given the same input. Definition A.3 formalizes the notion of *active* block/instruction and edge/incoming value. We write $\langle \ldots \rangle$ for ordered sequences.

Definition A.3 (Active Block/Edge). Let u and v be basic blocks in a program P. Block v is said to be *active* if its block condition is true. An instruction that belongs to v is *active* whenever block v itself is active. Similarly, an edge $u \longrightarrow v$ in P is said to be *active* if its edge condition is true. Finally, we say that an incoming value of a phi function is active whenever its corresponding incoming edge is active.

Lemma A.4 (Active Trace). *Let P_l be the partial linearization of P and let τ be the trace of blocks executed in P when given an arbitrary input. There exists a unique trace of blocks τ_l executed in P_l when given the same input such that, for every block $v \in \tau$, it follows that $v \in \tau_l$ and v is active in P_l.*

Proof. The proof will be by induction on τ:

Base case: In the base case, there exists a single block $v \in \tau$, which consequently is the first block of P. Since the first block has no predecessors in P and we do not add to nor remove any blocks from P_l — as stated by Lemma A.2 — the block condition of v is true. Therefore, block v will be active in P_l (see Definition A.3).

Induction step: Let $\tau = \langle v_1, \ldots, v_k \rangle$. By induction, we know that there exists τ_l executed in P_l such that v_i is active, $1 \leq i < k$. Let $v_j \in \tau$, $j < k$, be the predecessor of v_k that was executed in P. Since the edge $v_j \longrightarrow v_k$ was taken, we know that the edge condition of edge $v_j \longrightarrow v_k$ is true, and from the way block conditions are computed (Figures 10 and 12) — i.e. based on edge conditions — it follows that the block condition of v_k is true as well. Hence, we have that v_k is active in P_l. Furthermore, from Theorem A.1, we know that v_k will be reachable from v_j in P_l. Thus, since we are considering the same input for P and P_l, it must be that $v \in \tau_l$, which is the unique trace for such an input.

□

The next lemma is about the equivalence between the expressions in the original program P and the transformed program P'. It will be used in the proof for Theorem 4.13 (correctness). We write $[\![e]\!]$ to indicate the evaluation of e. When writing := in assignments, := can be either = (for general assignments) or <– (for stores).

Lemma A.5 (Expression Equivalence). *Let $T(P) = P'$ be the partial linearization of program P with instructions rewritten, as described in §§4.6 and 4.7. Let $x := e$ be an assignment in P and let $x' := e'$ be the counterpart assignment of x in P'. Then, for any instance of the inputs such that $x := e$ is executed in P, it follows that $x' := e'$ is active in P' and $[\![e]\!] = [\![e']\!]$.*

Proof. Consider an arbitrary instance of the inputs and let τ be the ordered sequence of assignments in P when given such an input. From Lemma A.4, any block executed in P is active in P'. Hence, if $x := e \in \tau$, its counterpart $x' := e'$ will be active in P', since they belong to the same block. It remains to show the correspondence between expressions e and e'. The proof will be by structural induction on the multiple assignment forms:

(a) $\mathtt{x}$ = **public** | **secret:** These special assignments only indicate that x corresponds to an input and the transformation described in this chapter never really touches them. Hence, if the assigment is active in P', the value assigned will be the same as in P, for we are considering the same inputs for both the original program P and the repaired version P'.

(b) $\mathtt{x} = e$**:** Let expression e be composed by subcomponents $v_1, \ldots, v_n$. Then, by induction on each v_i, $1 \le i \le n$, we know that $[\![v_i]\!] = [\![v'_i]\!]$, where v'_i is the counterpart of v_i in P'. Thus, it must be that e evaluates to the same value in both P and P', when given the same inputs.

(c) $\mathtt{x} = \mathtt{m}[i]$**:** From rule $rewrite_{ld}$ seen in Figure 20, we know that this assignment will be rewritten as $\mathtt{x} = \mathtt{a}[\mathtt{i'}]$. But, since the assignment is active, i.e. the block condition is true, it follows that $\mathtt{a} \equiv \mathtt{m}$ and $\mathtt{i'} \equiv i$. That is, the assignment that will be performed is the same in both P and P'. By induction we know that the values of $\mathtt{m}$ and i are the same in P and P', when given the same input. Hence, the memory access in P', under the described circumstances, will be the same as in P. It remains to show that the value stored in that address is the same in P and P', which follows by induction.

(d) $\mathtt{m}[i] \longleftarrow e$**:** From rule $rewrite_{st}$ (Figure 20), we know that this assignment will be rewritten as $\mathtt{a}[\mathtt{i'}] \longleftarrow e'$. Since the assignment is active in P', we have $e' \equiv e$, $\mathtt{a} \equiv \mathtt{m}$ and $\mathtt{i'} \equiv i$. By the same reasoning used in case (b), we can conclude that $[\![e]\!] = [\![e']\!]$. Thus, since the memory region accessed in P and P' will be the same, the value stored in this address, after the store operation is performed, will be the same in P and P'.

(e) $\mathtt{x}$ = **phi** $[e_1, \ell_1], \ldots, [e_n, \ell_n]$**:** First notice that there cannot be two edge conditions $EC(\ell_i \longrightarrow \ell)$ and $EC(\ell_j \longrightarrow \ell)$, $i \neq j$, that are true at the same time, since two edges cannot be taken simultaneously. From function $rewrite_\phi$ defined in Figure 19, we know that the edge conditions are used as the conditions of **ctsel** instructions that link incoming values that did not fit in the transformed phi function. The counterpart of $\mathtt{x}$ in P' will either be the transformed phi function or the last **ctsel** created, when any. Let $\ell_i \longrightarrow \ell$ be the active edge (i.e. edge condition is true). Then, either the incoming value e_i still is an incoming value in the transformed phi function, and consequently the condition of every **ctsel** will be false, or there will be a single **ctsel** whose condition is true and which will select e_i. In both cases, we have $[\![\textbf{phi}\ [e_1, \ell_1], \ldots, [e_n, \ell_n]]\!] = [\![e']\!]$.

(f) $\mathtt{x}$ = **ctsel** c, v_t, v_f**:** This kind of assignment is never modified by the transformation described in this chapter. Thus, it suffices to show that c, v_t and v_f evaluate to the same values in P and P', which follows directly by induction.

□

With Lemmas A.4 and A.5, we can prove Theorem 4.13:

Theorem 4.13 (Semantics). *Let $T(P) = P'$ be the partial linearization of program P with instructions rewritten, as described in §§4.6 and 4.7. If P' terminates, then programs P and P' produce the same set of side effects.*

Proof. The only instruction in our toy language that can produce side effects is a store. From function $rewrite_{st}$ defined in Figure 20, we have the following three cases:

Store is active: Let the store be of the form $m' <\!\!- e'$ and let $m <\!\!- e$ be its counterpart in P. Since $m' <\!\!- e'$ is active in P_l, the original memory region is accessed — i.e. $m \equiv m'$ — and the store updates the state of the memory. By Lemma A.5, we have $[\![e]\!] = [\![e']\!]$. Therefore, the state of m in both P and P_l is updated with the same value, thus causing the same effects.

Store is not active $\wedge$ access is safe: The original memory region is accessed, but the value to be assigned is replaced with the current value stored in that memory address; hence, the store is silent and no effect can be observed — i.e. the state of the memory does not change after the store is executed.

Store is not active $\wedge$ access is not safe: The original store is replaced with a store to **shadow** and the value to be assigned is replaced with the value currently stored in **shadow**; hence, the store is performed silently and no effect can be observed.

It remains to show that every store executed in program P is active in program P', which follows from Lemma A.4. □

The second property from Moll and Hack [2018] that we revisit is related to the post-dominance relation in G_l. We write $u \succeq^{PD} v$ for post dominance in the original graph G and $u \succeq^{PD}_l v$ for post dominance in the linearized graph G_l.

Lemma A.6 (PD for Deferral Edges [Moll and Hack 2018, Lemma B.3]). *Let $b \in Index$ be the block currently being visited by the loop at Lines 19–32 of the PCFL algorithm in Figure 5 and let s be a block such that $s \in T$ at Line 20 — in other words, s is the target of a deferral edge. Then, it follows that $s \succeq^{PD}_l b$.*

Lemma A.7 (PD for Tainted Branches). *Let u be a tainted block and let v_1 and v_2 be the successors of block u in program P. Then, after PCFL either $v_1 \succeq^{PD}_l v_2$ or $v_2 \succeq^{PD}_l v_1$.*

Proof. Notice that either $v_1 \longrightarrow v_2$ or $v_2 \longrightarrow v_1$ become a deferral edge at Line 31 of Figure 5, depending on which of them comes first in the compact ordering. Since the proof is the same for both cases, we will assume the first scenario: v_1 comes first in the compact ordering and thus $v_1 \longrightarrow v_2$ becomes a deferral edge. When node v_1 is visited, we have $(v_1, v_2) \in D$ and thus $v_2 \in T$ at Line 20 of Figure 5. Hence, from Lemma A.6 it follows that $v_2 \succeq^{PD}_l v_1$.

□

Lemma A.8 (PCFL Induces a Single Trace of Operations). *Let $\mathcal{I} = (\mathcal{S}, \mathcal{P})$ be the inputs taken by P, where $\mathcal{S}$ is the set of secret and $\mathcal{P}$ is the set of public inputs. Let u and v be blocks in P such that $v \succeq^{PD} u$ and assume that u is tainted. Then, after PCFL there is a single trace τ of operations from u to v for every instance of $\mathcal{P}$, regardless of $\mathcal{S}$.*

Proof. We shall assume an arbitrary fixed instance of $\mathcal{P}$. The proof will be by induction on the number of operation traces from block u to block v in the original program P:

Base case: If there is no tainted branch in the influence region of node u (see Def. 4.22), then for a fixed instance of $\mathcal{P}$ there are exactly two subtraces τ_1 and τ_2 of operations, each one starting with one of the successors w_1 and w_2 of u. Assuming that w_1 comes first than w_2 in the compact ordering (the proof is the same for the opposite scenario), we know that the edge $u \longrightarrow w_2$ will be removed and, from Lemma A.7, we know that $w_2 \succeq^{PD}_l w_1$. Hence, the instructions from w_2 to v will be merged into τ_1 — the trace in P that starts with w_1 — forming a single trace τ in P_l.

Induction step: Let W be the set of tainted blocks in the influence region of node u that are not further nested, i.e. blocks that have nesting level equals to the nesting level of u plus one. By induction, there is exactly one trace τ_w in P_l from every $w \in W$ to their post dominators. By combining these *disjoint* traces, following the paths in P_l, we get two traces τ_1 and τ_2 starting with each one of the successors of u. Then, the same reasoning that we used for the base case applies and we get a single trace τ from u to v in P_l.

□

Corollary A.9 (Local Operation Invariance). *Let $\mathcal{I} = (\mathcal{S}, \mathcal{P})$ be the inputs taken by P, where $\mathcal{S}$ is the set of secret and $\mathcal{P}$ is the set of public inputs. Let u and v be blocks in P such that $v \geq^{PD} u$ and assume that u is tainted. Then, after PCFL the region from block u to block v in P_l is operation invariant.*

Now we are ready to prove Theorem 4.16 that talks about operation invariance (see Def. 2.2), which we recall below:

Theorem 4.16 (PCFL Gives Operation Invariance). *Let P_l be the partial linearization of P. P_l is operation invariant.*

Proof. Let V be the set of tainted blocks with nesting level equals zero in the original program P (i.e. the outermost branches in P). From Lemma A.8, we know that, for a fixed instance of $\mathcal{P}$ (the set of public inputs), there is exactly one trace τ_v of operations in P_l for every $v \in V$. Hence, by joining these disjoint traces, we get a single trace τ for the entire program P_l. Therefore, for a fixed instance of $\mathcal{P}$, the sequence of instructions in P_l is the same regardless of $\mathcal{S}$. □

Corollary A.10 (Trace Relations). *Let P_l be the partial linearization of P. Let τ_1 and τ_2 be traces of memory addresses that correspond to the execution of P when given instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$, $\mathcal{S}_1 \neq \mathcal{S}_2$. If P_l is operation invariant, then $|\tau_1| = |\tau_2|$ and, for any i, the accesses to the addresses $\tau_1[i]$ and $\tau_2[i]$ are caused by the same instruction.*

We then move to the second property that characterizes isochronous programs: data invariance (see Definition 2.4). We restate Theorem 4.26 below and prove it right after.

Theorem 4.26 (The Data Contract). *Let $T(P) = P'$ be the partial linearization of P with loads and stores rewritten after Figure 20. If P is publicly safe, then P' is data invariant. If P is shadow safe, then either P' is data invariant or there exist two input instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$, $\mathcal{S}_1 \neq \mathcal{S}_2$, with corresponding traces of memory addresses τ_1 and τ_2 such that $\tau_1[i] =$ **shadow** or (exclusive) $\tau_2[i] =$ **shadow**, for some i.*

Proof. First, recall that, from Corollary A.10, it follows that $|\tau_1| = |\tau_2|$ and, for any i, $\tau_1[i]$ and $\tau_2[i]$ are memory accesses produced by the same instruction. Nevertheless, their addresses might not be equal. Let $\tau_1[i] = a$, $\tau_2[i] = a'$, and let $\mathtt{m}[i]$ be the combination of the base address and the index that caused such memory accesses. If P' is publicly safe, we know that $i < size(m)$. Hence, even if the access to $\mathtt{m}[i]$ is not active, the original address will always be selected in the **ctsel**s that define $\mathtt{m}$ and i ($rewrite_{st}$, Figure 20). Therefore, $a = a'$ and the theorem holds. If P' is not publicly safe, but is shadow safe, then let $\mathcal{S}_1$ and $\mathcal{S}_2$ be instances of the secret inputs such that $a \neq a'$ — otherwise, the theorem already holds. By inspecting rule $rewrite_{st}$, we know that the only possible values for $[\![\mathtt{m}[i]]\!]$ are the original address from P or **shadow**. Since we assumed $a \neq a'$, either address a or a' — but not both — must be the shadow memory. □

From Theorem 4.26, it follows that the only way to have **shadow** as one of the addresses accessed by P' is to have indices larger than the known size of the associated buffer.

A final property of partial control-flow linearization, as designed by Moll and Hack concerns optimality. In this case, optimality means that branches controlled by non-tainted predicates are not modified by linearization. Quoting from Moll and Hack: "*Partial linearization preserves uniform branches in blocks with uniform predicates, as implied by Theorem 4.1 [From Moll and Hack's work]*". To keep this paper self-sufficient, we restate optimality as a corollary of Theorem C.1 from Moll and Hack [2018]'s work:

Corollary 4.17 (Optimality). *Let ℓ be a basic block within a control-flow graph $G = (E, V)$, such that terminator(ℓ) = $\texttt{br}\,p, \ell_1, \ell_2$ and $\texttt{not}(tainted(p))$ is true. The edges $\ell \longrightarrow \ell_1$ and $\ell \longrightarrow \ell_2$ remain in E after partial control-flow linearization.*

Proof. Theorem C.1 in Moll and Hack's work states that given a dominance-compact block index, partial linearization will preserve an edge $b \longrightarrow y \in E$ if b is uniform (i.e., not tainted). This fact only yields Corollary 4.17. Yet, Theorem C.1 gives us more: $b \longrightarrow y$ remains also if there exists a block $d \in V$ with the following properties in G:

(1) d dominates the edge $b \longrightarrow y$
(2) edge $b \longrightarrow y$ is uniform in the dominance region d.

□