Memory-Safe Elimination of Side Channels

Luigi Soares[1], **Fernando Magno Quintão Pereira**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) — Brazil

{luigi.domenico,fernando}@dcc.ufmg.br

Abstract. *In this project, we find a new service for partial control-flow linearization (PCFL), a code transformation initially conceived to maximize work performed in vectorized programs. We show that PCFL can be employed as a defense mechanism 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 "break", "switch" and "continue", which are absent in the FaCT domain-specific language (2018). Like Constantine (2021), our code transformation ensures operation invariance, but without requiring profiling information. Additionally, in contrast to SC-Eliminator (2018), our implementation handles programs containing general, unbounded loops.*

1. Introduction

Side channels can be viewed as publicly observable outputs made available — usually unintentionally — from the execution of a computer system. They come in many flavors, e.g. runtime, cache hits and misses, power consumption and even sound! By observing the flow of information through side channels, adversaries can obtain sensitive information that should be otherwise protected. As such, side channels pose a major threat in today's world. Not surprisingly, news about the discovery of different types of attacks come out every year, as Figure 1 illustrates. In this project, we focus on *timing attacks*.

A program is said to be *isochronous* if its running time does not depend on sensitive information. *Isochronicity* is characterized by two properties: data and operation invariance. 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 [Reparaz et al. 2017, Almeida et al. 2016, Ngo et al. 2017, Barthe et al. 2019, Guarnieri et al. 2021] or to remove sources of time variance [Agat 2000, Almeida et al. 2020, Fell et al. 2019, Borrello et al. 2021, Cleemput et al. 2012, Van Cleemput et al. 2020, Gruss et al. 2017,

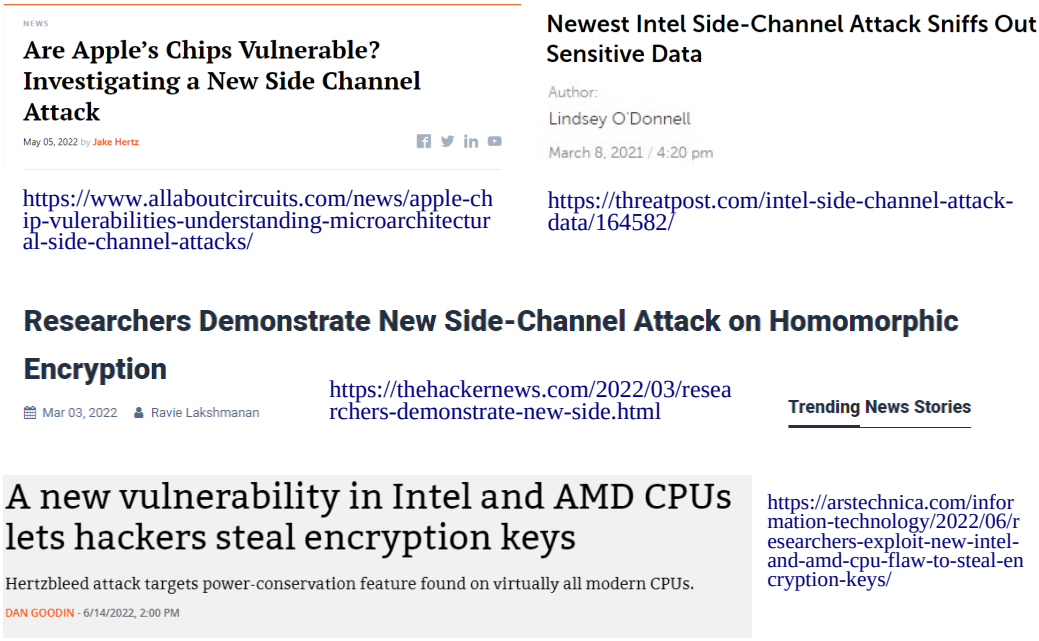

Figure 1. Recent news about the discovery of side-channel attacks.

Tizpaz-Niari et al. 2019, Chattopadhyay and Roychoudhury 2018, Wu et al. 2018]. And yet, the implementation of a static code transformation capable of removing time-based side channels from programs containing general loops remains an elusive endeavor.

Enter Partial Control-Flow Linearization. To solve this open problem, we repurpose Moll and Hack [2018]'s partial control-flow linearization (PCFL) algorithm, a code-optimization technique to speed up programs in the *Single-Instruction, Multiple-Data* (SIMD) model [Flynn 1972]. In a 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 said branches. The transformation keeps the uniform branches unchanged. In principle, PCFL bears as much importance to side-channel resistance as the fact that more kangaroos live in Australia than people in Uruguay.[1] However, replace "uniform" with *public* and "divergent" with *secret*, and *voilà*: we have a beautiful algorithm to produce isochronous programs!

1.1. Objectives

This project's main goal is to demonstrate the feasibility of adapting Moll and Hack [2018]'s partial control-flow linearization algorithm to protect programs against *operation-based* timing attacks. For that, we evaluated the following research questions:

- **RQ1:** By how much does the proposed approach increase code size?
- **RQ2:** What is the time taken to apply the proposed transformation onto programs?
- **RQ3:** How does the proposed approach impact the running time of programs?

[1] `https://twitter.com/redditcfb/status/1355288917558390786`

- **RQ4:** What are the security guarantees achieved by the proposed approach?
- **RQ5:** How the general C programs compiled with the proposed method compare with code written in a domain-specific language for constant-time cryptography?

1.2. Contributions

By leveraging Moll and Hack's partial control-flow linearization to erase secret-dependent branches, the following properties are delivered:

- **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 — 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 fed 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, because PCFL will disconnect the loop from the rest of the control-flow graph.

In this dissertation, we show that Moll and Hack's adapted algorithm makes programs operation invariant [Soares 2022, Thm. 5.2]. For a class of programs called as *publicly safe*, it also delivers data invariance [Soares 2022, Thm. 5.3] and, consequently, isochronicity [Soares 2022, Thm. 5.6]. Moreover, we demonstrate that code yielded by our transformation never leaks more than the original version [Soares 2022, Thm. 5.5]. Programs produced by our transformation still might have branches, as long as these branches are not influenced by secret data. Furthermore, our technique handles general loops (statically). Hence, we expand previous work in many ways:

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

```
1 // g (guess) and n public.
2 // pw (password) is secret.
3 int comp(int *g, int *pw, int n) {
4     for (int i = 0; i < n; i++)
5         if (g[i] != pw[i]) return 0;
6     return 1;
7 }
```

Figure 2. Function `comp` compares the user's guess `g` with a secret password `pw`. It returns immediately whenever two elements are different; as such, it is neither operation nor data invariant.

2. Partial Control-Flow Linearization by Example

To illustrate how Moll and Hack's partial control-flow linearization can be adapted to the context of side-channel resistance, consider the code from Figure 2. Function `comp` takes as input a password entered by the user and compares it against the secret password. It does that by comparing each character, one by one, returning immediatelly if a mismatch occurs. Suppose that the function was given the public inputs `g = {0, 0}` and `n = 2`. Then, the traces of instructions produced by the execution of the code when given the secret inputs `pw = {0,0}` and `pw = {1,0}` are, respectively,

$\tau_1^i =$ `(i = 0, i < n, g[i] != pw[i], i++, i < n, ret 1)` and
$\tau_2^i =$ `(i = 0, i < n, g[i] != pw[i], ret 0)`.

Notice that, by observing the sequence of instructions, it is possible to determine the exact position in which the user's guess and the actual password diverged. Therefore, function `comp` leaks information about the sensitive data, allowing an adversary to eventually discover it. One approach to prevent this type of attack is to remove branches whose conditions depend on sensitive information. For that, we can apply a modified version of Moll and Hack's partial control-flow linearization algorithm. The resulting code is shown in Figure 3. The code transformation employed can be broken as follows:

1. **Tainted flow analysis:** identifies which branches need to be linearized due to dependencies on sensitive information [Soares 2022, §5.2].
2. **Array-bounds analysis:** finds out symbolic bounds for arrays, so that we can decide if it is safe to perform memory accesses [Sperle Campos et al. 2016, §3.2].
3. **Rewrite interfaces:** wraps each pointer argument into a struct storing its content and size, which was inferred in the previous step.
4. **Predication analysis:** finds out which conditions control the execution of each block and edge of the program's control-flow graph [Soares 2022, §5.3].
5. **Partial Linearization:** linearizes the programs's control-flow graph, adapting Moll and Hack [2018]'s algorithm [Soares 2022, §§5.1 and 5.4.1].
6. **Rewrite code:** rewrites instructions to ensure that the code is correct and executable [Soares 2022, §§5.4.2 and 5.4.3]. This includes the creation of a so-called *shadow memory* that will be used as a surrogate address in order to prevent the introduction of out-of-bounds memory accesses [Soares and Pereira 2021].

Function `comp_pcfl`, depicted in Figure 3, is *operation invariant*. That is, the evaluation of `comp_pcfl` on an arbitrary public input leads to the same trace of instructions, regardless of the secret inputs fed into it. This is because the secret-dependent

```
1  typedef struct ptr_int_wrapped {
2      int *data; // the object's content
3      int size; // the object's inferred size
4  } ptr_int_wrapped;
5
6  // g (guess) and n are public, pw (password) is secret.
7  int comp_pcfl(ptr_int_wrapped *g, ptr_int_wrapped *pw, int n) {
8      int *shadow = (int *) malloc(sizeof(int));
9      int r = 1;
10     int loop_cond = 1;
11     for (int i = 0; i < n; i++) {
12         int *g_addr = ctsel(loop_cond | (i < g->size), g->data, shadow);
13         int *pw_addr = ctsel(loop_cond | (i < pw->size), pw->data, shadow);
14         int g_idx = ctsel(loop_cond | (i < g->size), i, 0);
15         int pw_idx = ctsel(loop_cond | (i < pw->size), i, 0);
16         int g_i = g_addr[g_idx];
17         int pw_i = pw_addr[pw_idx];
18         loop_cond &= g_i == pw_i;
19         r &= loop_cond;
20     }
21     return r;
22 }
```

Figure 3. Partially linearized version of the code seen in Figure 2.

branch at Line 5 of the Figure 2 was removed. In Figure 3, the loop can only terminate due to the public input `n`. Function `comp_pcfl`, however, is not necessarily *data invariant* (and, consequently, not necessarily *isochronous*). To see that, suppose that `comp_pcfl` was called with the public inputs `g->data` $= \{0, 0\}$ and `n = 2`. Suppose, further, that the inference algorithm was not capable of determining the size of the array, assigning to it a size of zero. Then, the traces of memory addresses produced by the execution of the code when given the secret inputs `pw` $= \{0, 0\}$ and `pw` $= \{1, 0\}$ are, respectively,

$\tau^a_{l,1}$ = `(g->data[0], pw->data[0], g->data[1], pw->data[1])` and
$\tau^a_{l,2}$ = `(g->data[0], pw->data[0], shadow, shadow)`.

If, however, the array-bounds analysis had correctly estimated the real sizes of the arrays `g` and `pw`, then the traces of memory addresses would be identical, thus characterizing data invariance. There is a class of programs for whom PCFL always ensures data invariance: the *publicly-safe* programs [Cauligi et al. 2019]. In the words of Cauligi et al.: "for a program to be amenable to constant-time compilation, the source must be publicly safe: it must be free from buffer overflows and undefined behavior using only public-visible information, i.e. the code must be safe even after removal of secret-dependent control-flow". `FaCT` programs are, by design, publicly safe. Nevertheless, public safety can be achieved in C-like code by requiring that (i) memory is indexed only by public data and (ii) memory accesses are proven to be in-bounds. Under these assumptions, the programs resulted from our transformation meet the same guarantees delivered by `FaCT`.

3. Results

We implemented our ideas on top of LLVM 13.0 [Lattner and Adve 2004]. We initially developed a loop-free transformation (`Lif`) [Soares and Pereira 2021] capable

of repairing programs whose loops can be completely unrolled. The final prototype that incorporates Moll and Hack's partial control-flow linearization algorithm, and deals with general loops, was built on top of Lif. We refer to this implementation as `PCFL`. We compared both methods with `SC-Eliminator` [Wu et al. 2018], `Constantine` [Borrello et al. 2021] and `FaCT` [Cauligi et al. 2019].

3.1. RQ1, RQ2 and RQ3: Code Size, Transformation Time and Performance

Table 1 summarizes the results obtained in regard to the nine benchmarks that `PCFL`, `Lif`, `SC-Eliminator` and `Constantine` can handle. Notice that code size for `Lif` and `SC-Eliminator` is much bigger because these tools require that loops are fully unrolled. In relative terms, with respect to the nine benchmarks that the four tools can handle, our PCFL implementation increased code size by $1.02\times$ and running time by $1.61\times$, while being the fastest transformation.

Table 1. Summary of results with regard to the nine benchmarks that `PCFL`, `Lif`, `Constantine` and `SC-Eliminator` can handle. Numbers are arithmetic means. Measurements happen after programs are transformed and then optimized with `LLVM opt -O3`. `Original` refers to the benchmark without any transformation. `PCFL` and `Lif` correspond to our implementations. `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 approaches only do control-flow linearization, but achieves data invariance for publicly-safe programs.

Tool	Original	PCFL	Lif	CTT-Orig	CTT-CFL	SC-Orig	SC-CFL
Size (#LLVM instrs.)	330.78	337.00	15,342.11	464.67	365.89	10,863.56	8,444.89
Running time (μs)	3.23	5.21	12.09	14.94	6.27	5.19	4.60
Linearization time (ms)		33.49	271.61	2,045.22	65.19	2,697.06	2,149.28

3.2. RQ4: Security Evaluation

Operation Invariance. To verify that PCFL delivers operation invariance, we relied on `CTGrind` [Langley 2010], a `Valgrind` plugin that determines if a program contains a branch that reads data tainted by secret information. As Figure 4 demonstrates, we certified empirically that our implementation of PCFL transformed all the 13 benchmarks into operation-invariant programs.

Data Invariance. When probing data variance in Figure 4, we used an LLVM instrumentation pass to verify if the sequence of addresses accessed by each kernel is the same, regardless of the input. In several cases, we observe that neither `SC-Eliminator` nor `Constantine` achieve data invariance. This apparent failure is a consequence of the threat model that these tools assume: they consider all the memory accesses to a cache line as the same access. Our failures to achieve complete data invariance, in turn, are due to the fact that some memory accesses are replaced with the shadow memory. Nevertheless, we could verify that data invariance holds for publicly-safe programs. In particular, we ported `ssl3` and `donna` from `FaCT`. When written in `FaCT`, every benchmark is publicly safe, and we succeeded in delivering isochronicity for these programs.

	Original			PCFL				Lif				SC-Eliminator				Constantine			
	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3
hash-one	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N	N
plain-many	N	N	N	Y	N	Y	Y	X	X	X	X	X	X	X	X	X	X	X	X
plain-one	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	N
donna	N	N	N	Y	Y	Y	Y	UL	UL	UL	UL	UL	UL	UL	UL	Y	N	N	N
ssl3	Y	N	N	Y	Y	Y	Y	UL	UL	UL	UL	UL	UL	UL	UL	N	N	N	N
log-rdct	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	N
3way	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
des	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y
loki91	Y	N	N	Y	Y	Y	Y	UL	UL	UL	UL	UL	UL	UL	UL	Y	Y	Y	Y
cast5	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	Y	Y	Y
dijkstra	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	Y	N	N
findmax	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
histogram	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	Y	Y	N	N	N

N Property not verified Y Property verified UL Unbounded loop X Tool crashes or transformed program crashes

Figure 4. Security guarantees achieved by the different tools. `Cor` indicates if the transformed program produces the same output as its original counterpart (i.e. if the transformed program is *correct*). `Data` refers to data invariance. `Opr` refers to operation invariance without compiler optimizations. `Opr3` refers to operation invariance at the `LLVM opt -O3` optimization level.

3.3. 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]. As Figure 5 shows, the programs written in `FaCT` are, usually, shorter and faster. However, the programs that we generate contain code to ensure memory safety. Therefore, every linearized load and store operation in a program produced with our technique will contain extra instructions absent in the equivalent `FaCT` program. In `FaCT`, developers use a type qualifier, `assume`(e), which let them specify that expression e is the upper bound of an array. This clause works as a contract: the compiler does not generate code to ensure in-bounds 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.

	PCFL					FaCT				
	Time (µs)	.o	Instrs.	.o	Instrs.	Time (µs)	.o	Instrs.	.o	Instrs.
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
		without main		with main			without main		with main	

Figure 5. 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.

4. Related Work

This project 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 — a code transformation that converts control dependencies into data dependencies — inasmuch as it spares uniform branches from being predicated. 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 Hacks'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. 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 work is not about the design of a partial control-flow linearization approach. We reuse Moll and Hacks's algorithm almost without modifications. There exists 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, linearized loops only terminate when all threads exit it. In our case, a loop can have multiple exits: any exit that is only dependent on public data will be left untouched by our transformation.

Side-Channel Elimination via Control-Flow Linearization. The literature contains many examples of protections against such attacks. This dissertation is concerned with the so-called *white-box* mitigations, which require intervening in the software. The current state-of-the-art control-flow linearization technique to make programs written in a general-purpose language operation invariant is due to Borrello et al. [2021]. In Borrello et al.'s implementation, loops are linearized *just-in-time* by replacing the normal trip count of a loop with a special induction variable that dictates how many times that loop should execute. This kind of transformation requires loop profiling in order to identify the number of iterations that the loop performs, which implies an important limitation: it is undecidable to find inputs to exercise specific parts of a program's code, as Rice's Theorem indicates [Rice 1953]. By repurposing Moll and Hack [2018]'s partial control-flow linearization algorithm, we show that it is possible to yield operation-invariant programs without the need of a dynamic analysis.

5. 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. In contrast to previous techniques [Wu et al. 2018], our approach is capable of transforming programs with unbounded loops, and does not require profiling information [Borrello et al. 2021].

Furthermore, code generate by our tools are competitive when compared to previous work [Wu et al. 2018, Borrello et al. 2021, Cauligi et al. 2019], and the transformation itself has practical compilation time.

References

Agat, J. (2000). Transforming out timing leaks. In *POPL*, page 40–53, New York, NY, USA. Association for Computing Machinery.

Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016). Verifying constant-time implementations. In *SEC*, page 53–70, USA. USENIX Association.

Almeida, J. B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., and Strub, P. (2020). The last mile: High-assurance and high-speed cryptographic implementations. In *Security & Privacy*, pages 965–982, New York, NY, USA. IEEE.

Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., and Trieu, A. (2019). Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL).

Borrello, P., D'Elia, D. C., Querzoni, L., and Giuffrida, C. (2021). Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS*, page 715–733, New York, NY, USA. Association for Computing Machinery.

Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R. S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., and Stefan, D. (2019). Fact: A dsl for timing-sensitive computation. In *PLDI*, page 174–189, New York, NY, USA. Association for Computing Machinery.

Chattopadhyay, S. and Roychoudhury, A. (2018). Symbolic verification of cache side-channel freedom. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2812–2823.

Cleemput, J. V., Coppens, B., and De Sutter, B. (2012). Compiler mitigations for time attacks on modern x86 processors. *Trans. Archit. Code Optim.*, 8(4).

Fell, A., Pham, H. T., and Lam, S.-K. (2019). Tad: Time side-channel attack defense of obfuscated source code. In *ASP-DAC*, page 58–63, New York, NY, USA. Association for Computing Machinery.

Ferrante, J. and Mace, M. (1985). On linearizing parallel code. In *PLDI*, page 179–190, New York, NY, USA. Association for Computing Machinery.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *Transactions on Computers*, 21(9):948–960.

Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. (2017). Strong and efficient cache side-channel protection using hardware transactional memory. In *SEC*, page 217–233, USA. USENIX Association.

Guarnieri, M., Köpf, B., Reineke, J., and Vila, P. (2021). Hardware-software contracts for secure speculation. In *Security & Privacy*, pages 1868–1883, New York, US. IEEE.

Karrenberg, R. and Hack, S. (2012). Improving performance of opencl on cpus. In *CC*, page 1–20, Berlin, Heidelberg. Springer-Verlag.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, page 104–113, Berlin, Heidelberg. Springer-Verlag.

Langley, A. (2010). Ctgrind—checking that functions are constant time with valgrind.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, page 75, Washington, USA. IEEE Computer Society.

Moll, S. and Hack, S. (2018). Partial control-flow linearization. In *PLDI*, page 543–556, New York, NY, USA. Association for Computing Machinery.

Moreira, R. E., Collange, C., and Quintão Pereira, F. M. (2017). Function call re-vectorization. In *PPoPP*, page 313–326, New York, NY, USA. Association for Computing Machinery.

Ngo, V. C., Dehesa-Azuara, M., Fredrikson, M., and Hoffmann, J. (2017). Verifying and synthesizing constant-resource implementations with types. In *Security and Privacy*, pages 710–728, Washington, DC, USA. IEEE.

Reparaz, O., Balasch, J., and Verbauwhede, I. (2017). Dude, is my code constant time? In *DATE*, page 1701–1706, Leuven, BEL. European Design and Automation Association.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366.

Soares, L. and Pereira, F. M. Q. a. (2021). Memory-safe elimination of side channels. In *CGO*, pages 200–210, Washington, USA. IEEE.

Soares, L. D. C. (2022). Memory-safe elimination of side-channels. Master's thesis, UFMG, Computer Science Department. Online available at: `http://hdl.handle.net/1843/42564`.

Sperle Campos, V. H., Alves, P. R., Nazaré Santos, H., and Quintão Pereira, F. M. (2016). Restrictification of function arguments. In *CC*, page 163–173, New York, NY, USA. Association for Computing Machinery.

Tizpaz-Niari, S., Černý, P., and Trivedi, A. (2019). Quantitative mitigation of timing side channels. In *CAV*, pages 140–160, Heidelberg, Germany. Springer.

Van Cleemput, J., De Sutter, B., and De Bosschere, K. (2020). Adaptive compiler strategies for mitigating timing side channel attacks. *Transactions on Dependable and Secure Computing*, 17(1):35–49.

Wu, M., Guo, S., Schaumont, P., and Wang, C. (2018). Eliminating timing side-channel leaks using program repair. In *ISSTA*, page 15–26, New York, NY, USA. Association for Computing Machinery.