Formal Hardware/Software Models for Cache Locking Enabling Fast and Secure Code

Hatchikian-Houdot Jean-Loup, Pierre Wilke, Frédéric Besson, Guillaume Hiet

Formal Hardware/Software Models for Cache Locking Enabling Fast and Secure Code

Jean-Loup Hatchikian-Houdot[1], Pierre Wilke[2], Frédéric Besson[1], and Guillaume Hiet[2]

[1] Inria, Université de Rennes, CNRS, IRISA, France
[2] CentraleSupélec, Inria, CNRS, IRISA, Univerité de Rennes, France

Abstract. Constant-time programming is the *de facto* standard to protect security-sensitive software against cache-based timing attacks. This software countermeasure is effective but may incur a significant performance overhead and require a substantial rewrite of the code. In this work, we propose a secure cache-locking hardware mechanism which eases the writing of secure code and has little execution overhead. To reason about the security of software, we propose a high-level leakage model such that accesses to locked memory addresses do not generate any observable leakage. To ensure the adequacy of this leakage model, we also propose a concrete hardware leakage model for a RISC-V micro-controller where the secure code may be interrupted, at any time, by some arbitrary malicious code. Using the Observational Non-Interference setting, we show formally that the security of the software model is preserved at the hardware level. We evaluate the effectiveness and performance of this mechanism, notably on block ciphers. We also propose and evaluate a new constant-time sorting algorithm.

1 Introduction

In the Internet of Things context, there is an explosion in the number of small devices based on low-end microcontrollers. These devices have restricted resources (computing power, battery, memory) yet embed secrets, *e.g.* secret keys, that need to be protected. Protecting assets under such strict resource constraints is challenging, yet, opens the opportunity to revisit software/hardware security countermeasures. Beyond logic attacks, *e.g.* exploitation of buffer overflows, an important class of attacks are passive side-channel attacks whereby an attacker measures some physical properties of a computation such as power consumption, electromagnetic emanations or timing, in order to deduce confidential information. The paper focuses on cache-based timing attacks that are among the easiest to exploit [26, 28, 24] since they do not require any physical access to the device. Device vendors are increasingly concerned about these attacks since many IoT devices both manipulate confidential data and allow the execution of code under the control of a possible attacker. We show that cache locking is an efficient hardware mechanism which can be exposed at the ISA level using a hardware/-software contract [17] and can be leveraged to ease the writing of secure code

free of timing leakage. Pure hardware countermeasures have been proposed. However, static cache partitioning [14] limits the available size of the cache during the whole execution and cache randomisation [25] necessitates a robust random generator, which is difficult to achieve with limited hardware resources. Our solution has the advantage of locking only the necessary amount of cache lines when needed, and requires few modifications to the hardware. Moreover, compared to a software countermeasure such as the (cryptographic) constant-time programming discipline [8, 23, 21], we enable a more flexible programming discipline lifting the constraint forbidding memory accesses with security sensitive data [21].

Our cache locking hardware/software contract is expressed as a high-level leakage model that a programmer can use to reason about the security of its code and is preserved by the hardware implementation of the cache-locking mechanism. Our contributions are phrased as follows:

- a high-level software leakage model which relaxes the usual constant-time model so that locked memory accesses are not observable;
- a systematic security study of design choices for cache/memory consistency schemes (*write through* or *write back*, *write around* or *write allocate*); their security implication and leakage model;
- a formal proof that a secure code according to the high-level software leakage model is still secure for a concrete hardware leakage model accommodating for an arbitrary attacker sharing the cache;
- experimental evidence that cache-locking enables fast and secure implementations of block ciphers, *e.g.* AES, and a variant of Batcher's sort.

The rest of the paper is organised as follows. In Section 2, we present our hardware and attacker model, recall the main features of the cache-locking mechanism of Gaudin *et al.* [16] and the definition of Observational Non-Interference that will serve as our security property. In Section 3, we present our formal model of a memory equipped with a cache and a locking mechanism. Section 4 presents our experimental results demonstrating that, using our cache-locking mechanism, it is feasible and simple to get faster secure implementations for standard algorithms. In Section 5, we prove that the ONI software contract is preserved at the hardware level. Related work is presented in Section 6 and Section 7 concludes. The benchmarks and a Coq development with mechanised proofs are available[3].

2 Hypotheses and Background

Hypotheses and Attacker Model. Our target hardware is a single-core RISC-V microcontroller without out-of-order execution or speculation. Nonetheless, it features a shared cache for code and data. We make the pessimistic assumption that the secure program may be interrupted at any time by some arbitrary attacker sharing the memory cache and able to observe all the memory accesses.

[3] `https://gitlab.inria.fr/scratchs-public/esorics2024-artefact`

Gaudin et al. *[16]* propose a cache locking mechanism inspired by PLcache [27]. They implement a 4-way set-associative data cache with a LRU eviction policy. For a given cache set, 3 out of the 4 cache lines may be marked as locked. The locking mechanism is exposed at the ISA level by a `lock` and a `unlock` instruction which are used to toggle the lock bit of a cache line. A locked cache line is loaded into the cache and cannot be evicted unless it is explicitly unlocked. This ensures that accessing a locked address is necessarily a cache hit.

Observational Non-Interference (ONI) is the software/hardware contract that we ensure for the cache-locking mechanism studied in this paper. Informally, two executions are ONI if, starting from initial configurations that only differ on the secret values, their leakage traces are identical. In other words, the executions are indistinguishable for an attacker observing the leakage trace. Our presentation follows Barthe *et al.*, [8] where they formalise secure compilation as the preservation of ONI. The program execution is modelled by a small-step operational semantics which is augmented by a leakage trace. The initial configuration is constructed from a set $\mathcal{I}$ of inputs.

Definition 1 (Program Model). *Given program states State, trace events Event and input $\mathcal{I}$, a program P is given by a pair $(I, \cdot \xrightarrow{t} \cdot)$ where:*

- *$I : \mathcal{I} \to \mathcal{P}(State)$ constructs an initial configuration from an input $i \in \mathcal{I}$*
- *$\cdot \xrightarrow{t} \cdot$ is the deterministic transition relation of the program which emits a trace $t \in Event^*$.*

We write $\mathcal{F}$ for the program states without successors: $\mathcal{F} = \{a \mid \neg\exists b, t. a \xrightarrow{t} b\}$.

The leakage traces are concatenated along the execution. Therefore, the multi-step relation of a relation $\cdot \xrightarrow{t} \cdot$ is defined as follows:

$$\frac{}{a \xrightarrow{\epsilon}{}^{0} a} \qquad \frac{a \xrightarrow{t} a' \quad a' \xrightarrow{t'}{}^{n} a''}{a \xrightarrow{t \cdot t'}{}^{n+1} a''}$$

We also consider an equivalence relation $\phi \subseteq State \times State$ over the initial states which identifies initial configurations of a program that are deemed indistinguishable *i.e.* only differ by the value of secret data.

Definition 2. *(ONI) A program $P = (I, \cdot \xrightarrow{t} \cdot)$ is observationally non-interferent for a predicate ϕ, written $ONI(\phi, P)$, if and only if the following holds:*

$$\bigwedge \left\{ \begin{array}{l} a \in I(i) \wedge a' \in I(i') \wedge \phi(a, a') \\ a \xrightarrow{t}{}^{n} b \wedge a' \xrightarrow{t'}{}^{n} b' \end{array} \right\} \implies t = t' \wedge (b \in \mathcal{F}(P) \iff b' \in \mathcal{F}(P))$$

Barthe *et al.* [8] propose reasoning principles to show that Observational Non-Interference is preserved by compiler passes.

In Section 5, we propose a reasoning principle adapted to a setting where the secure program shares resources (*e.g.*, a memory cache) and the CPU with an arbitrary attacker.

3 Memory Interface and Models of Cache

We provide an abstract model of a memory sub-system made of a cache and a main memory. The model is instantiated by a software model where the cache only records locked addresses and several concrete secure hardware models for cache policies such as *write-allocate*, *write-around* or *write-back*. The memory is modelled by a relation of the form

$$\rho \vdash_{id} (c, m) \xrightarrow{\tau, v} (c', m')$$

where $\rho \in \texttt{REQ}$ is a memory request; $id \in \{\mathcal{D}, \mathcal{A}\}$ is the process identifier; $c, c' \in \texttt{CACHE}$ are models of the cache; $m, m' \in \texttt{MEMORY} = \texttt{ADDRESS} \to \texttt{VALUE}$ represent the main memory; $v \in \texttt{VALUE}_\perp$ is the (optional) answer that is returned to the process performing the request and τ is the leakage that is associated to the request. Typically, at the hardware level, the leakage is either a cache hit (fast) or a cache miss (slow) that is observable by an attacker. The memory requests are as follows: $\texttt{REQ} \ni \rho ::= \texttt{load}(a) \mid \texttt{store}(a, v) \mid \texttt{lock}(a) \mid \texttt{unlock}(a)$ where v represents a value and a is an address. An address a can be uniquely decomposed into a triple $(t, s, w) \in \texttt{TAG} \times \texttt{SETID} \times \texttt{WORDID}$ where s identifies the cache set where the address a may be cached, w identifies the word in the cache line and the t is the tag that is stored in the cache. We define $tag\&set(t, s, w) = (t, s)$. The request is only performed if the address is in the memory space of the process (written $a \in id$). The $\texttt{lock}(a)$ and $\texttt{unlock}(a)$ instructions lock or unlock the address a. The `lock` instruction leaks information. However, its implementation ensures that subsequent `load` and `store` requests at a locked address are indistinguishable. The `load` returns, as value v, the value read at address a. Depending on the cache state, it is either read from cache or memory.

3.1 Software Cache Model

The software cache model provides a high-level security model that is independent of the concrete implementation and allows reasoning about the security of the program in isolation, *i.e.* without considering an explicit attacker. Our model refines the usual constant-time model to account for the `lock` and `unlock` primitives. Without `lock` instructions, our model coincides with the usual constant-time model [8] where all the memory accesses are leaked. The software cache model $L \in \texttt{CACHE}_\mathbf{S}$ only records the cache lines that are locked. As caches provide a number of ways for each cache set, a convenient representation is:

$$\texttt{CACHE}_\mathbf{S} = \texttt{SETID} \to \mathcal{P}(\texttt{TAG})$$

The transition relation is given in Fig. 1. For `load` and `store`, we leak the trace $\tau = \mathcal{L}^\mathbf{S}(a, a \in L)$ *i.e.* $tag\&set(a)$ if the address a is not locked. The `lock` and `unlock` operations always leak $tag\&set(a)$. For a $\texttt{load}(a)$ request, the address is fetched from the memory M and $M(a)$ is returned to the calling process. The other requests return nothing ($\perp$). The `lock` request checks that the address can be locked in the cache. This is the case if the address is already locked or the cache still has a free way in the cache set.

$$\frac{a \in id \qquad \tau = \mathcal{L}^{\mathbf{S}}(a, a \in L)}{\texttt{load}(a) \vdash_{id} (L, M) \xrightarrow[\mathbf{S}]{\tau, M(a)} (L, M)} \qquad \frac{a \in id \qquad \tau = \mathcal{L}^{\mathbf{S}}(a, a \in L)}{\texttt{store}(a, v) \vdash_{id} (L, M) \xrightarrow[\mathbf{S}]{\tau, \bot} (L, M[a \mapsto v])}$$

$$\frac{\begin{array}{c} a \in id \qquad canLock(a, L) \\ \tau = tag\&set(a) \end{array}}{\texttt{lock}(a) \vdash_{id} (L, M) \xrightarrow[\mathbf{S}]{\tau, \bot} (L \cup \{a\}, M)} \qquad \frac{a \in id \qquad \tau = tag\&set(a)}{\texttt{unlock}(a) \vdash_{id} (L, M) \xrightarrow[\mathbf{S}]{\tau, \bot} (L \setminus \{a\}, M)}$$

Given an address $a = (t, s, w)$, we have:

$$\begin{array}{l} a \in L \triangleq t \in L(s) \qquad canLock(a, L) \triangleq a \in L \vee |L(s)| < nbways{-}1 \\ L \cup a \triangleq L[s \mapsto L(s) \cup \{t\}] \qquad L \setminus a \triangleq L[s \mapsto L(s) \setminus \{t\}] \\ \mathcal{L}^{\mathbf{S}}(a, b) = \textbf{if } b \textbf{ then } \bullet \textbf{ else } tag\&set(a) \qquad tag\&set(a) = (t, s) \end{array}$$

Fig. 1. Software cache semantics

Example 1. Consider the following code which applies a permutation given by `perm` to the secret buffer `sec`. If the array `perm` is not locked, the trace of memory (`load`) events is $\prod_{i=0}^{n}(tag\&set(\texttt{sec} + i); tag\&set(\texttt{perm} + \texttt{sec}[i]))$. The program is not secure because the load events depend on the secret value $\texttt{sec}[i]$.

```
lock_array(perm,n);
for(i = 0; i<n; i++)
 sec[i] = perm[sec[i]];
unlock_array(perm,n);
```

```
ct_load(t, i, n){
 res = 0;
 for(j = 0; j < n; j++)
  res = ct_select(i==j, t[j], res);
 return res;}
```

A classic secure countermeasure may replace `perm[sec[i]]` by `ct_load(perm, sec[i],n)` where `ct_load` is a constant-time load gadget which iterates over all the indices of `perm` and performs a constant-time selection over the desired index (`ct_select(cond,a,b)` returns `a` if `cond` is true, `b` otherwise). For our example, this turns a vulnerable $\mathcal{O}(n)$ code into a $\mathcal{O}(n^2)$ secure code. If the `perm` array is locked, the situation is quite different. For any secret array `sec`, we get the exact same trace of load events: $\prod_{i=0}^{n}(tag\&set(\texttt{sec} + i); \bullet)$ and therefore, the code is secure, for the software model, while retaining the $\mathcal{O}(n)$ complexity.

3.2 Hardware Cache Models

In this part, we present our hardare model for a n-way associative cache, a minimal set of primitive for implementing classic caching policies, and discuss secure guidelines for using them.

Hardware Cache Model Unlike the software model, a hardware cache contains concrete cache lines. We model a standard n-way associative cache where the eviction policy `USAGE` is local to a cache set and a cache line is made of

$$
\begin{array}{ll}
\mathit{fetch} & : \mathtt{CACHE_H} \times \mathtt{MEMORY} \times \mathtt{ADDRESS} \times \mathtt{WAY} \rightarrow^{\mathtt{LEAK}} \mathtt{CACHE_H} \\
\mathit{write\_back} & : \mathtt{CACHE_H} \times \mathtt{MEMORY} \times \mathtt{ADDRESS} \times \mathtt{WAY} \rightarrow^{\mathtt{LEAK}} \mathtt{MEMORY} \\
\mathit{update\_usage} & : \mathtt{CACHE_H} \times \mathtt{REQ} \rightarrow \mathtt{CACHE_H} \\
\mathit{lock\_way} & : \mathtt{CACHE_H} \times \mathtt{ADDRESS} \times \mathtt{WAY} \rightarrow \mathtt{LOCKED} \rightarrow \mathtt{CACHE_H} \\
\mathit{is\_locked} & : \mathtt{CACHE_H} \times \mathtt{ADDRESS} \rightarrow \mathtt{LOCKED} \\
\mathit{write\_in\_cache} & : \mathtt{CACHE_H} \times \mathtt{ADDRESS} \times \mathtt{WAY} \times \mathtt{VALUE} \rightarrow \mathtt{CACHE_H} \\
\mathit{read\_from\_cache} & : \mathtt{CACHE_H} \times \mathtt{ADDRESS} \times \mathtt{WAY} \rightarrow \mathtt{VALUE} \\
\mathit{way\_of\_address} & : \mathtt{CACHE_H} \times \mathtt{ADDRESS} \rightarrow \mathtt{WAY}_{\perp} \\
\mathit{way\_to\_evict} & : \mathtt{CACHE_H} \times \mathtt{REQ}\times \rightarrow (\mathtt{ADDRESS} \times \mathtt{WAY})_{\perp}
\end{array}
$$

Fig. 2. Interface of Hardware Caches

i) a *valid* bit indicating whether the line holds valid data; ii) a *dirty* bit indicating whether the content of the cache line needs to be synchronised with the main memory; iii) a *locked* bit indicating whether the cache line is locked; iv) a memory address tag; and v) the content of the cache line.

$$
\begin{array}{ll}
\mathtt{CACHELINE} & \triangleq \mathtt{VALID} \times \mathtt{DIRTY} \times \mathtt{LOCKED} \times \mathtt{TAG} \times \mathtt{VALUE}^{nb\_words} \\
\mathtt{CACHE_H} & \triangleq \mathtt{SETID} \rightarrow \mathtt{USAGE} \times \mathtt{CACHELINE}^{n}
\end{array}
$$

Each access to the memory interface generates a leakage trace made of the following hardware events: $\mathtt{LEAK_H} ::= \mathbf{F}(t,s) \mid \mathbf{WB}(t,s) \mid \mathbf{Hit}$ where $\mathbf{F}(t,s)$ is leaked when a cache line is fetched from memory; $\mathbf{WB}(t,s)$ is leaked when a cache line is written back to memory and **Hit** corresponds to a cache hit. The $\mathbf{F}(t,s)$ and $\mathbf{WB}(t,s)$ events are generated in case of cache misses. They model our assumption that RAM memory accesses may leak information, for instance, in case of contention over the RAM controller.

Hardware Cache Interface The memory requests *i.e.* `load`, `store`, `lock` and `unlock`, are implemented using the interface of Fig. 2. Function $\mathit{fetch}(c,m,a,w)$ updates the cache c with the content of the memory m at address a. It is stored in the way w of the cache set associated with the address a. Function $\mathit{write\_back}(c,m,a,w)$ synchronises the content of the memory m with the cache line for the address a that is stored in the way w, if that line is dirty. Similarly to *fetch*, *write_back* may leak the event $\mathbf{WB}(\mathit{tag}\&\mathit{set}(a))$. Function $\mathit{update\_usage}(c,r)$ updates the eviction policy depending on the memory request r. Function $\mathit{lock\_way}(c,a,w,b)$ sets to the boolean b, the lock status of the way w of the cache set of the address a. It is used by the `lock` and the `unlock` requests. Function $\mathit{is\_locked}(c,a)$ tells whether there is a way of the cache set of address a such that the address a is locked. Function $\mathit{write\_in\_cache}(c,a,w,v)$ is used by the `store` request. It writes the value v in the cache set of the address a in the way w. Function $\mathit{read\_from\_cache}(c,a,w)$ returns the value v that is stored in the cache set of the address a in the way w. It is used when the way w is caching the content of the address a. Function $\mathit{way\_of\_address}(c,a)$ returns the way where the content of the address a is cached; or $\perp$ if the address is not

cached. Finally, the function $\mathit{way\_to\_evict}(c, r)$ assumes that the memory request r accesses an address a, and returns a way w' that can be evicted together with the address a' that is currently cached in way w'. The decision is based on the current eviction policy of the cache set of a and the current request r. For a $\texttt{lock}(a)$ request, it may return $\bot$ if all the available ways are already locked.

The following inference rule illustrates how the hardware interface can be used to implement a `store` request using a *write-back* strategy.

$$\frac{\begin{array}{cc} & a \in id \\ \mathit{way\_of\_address}(C, a) = \bot & \mathit{way\_to\_evict}(C, \texttt{store}(a, v)) = \lfloor a', w \rfloor \\ \mathit{write\_back}(C, M, a', w) =^{\alpha} M' & \mathit{update\_usage}(C, \texttt{store}(a, v)) = C_1 \\ \mathit{fetch}(C_1, M', a, w) =^{\beta} C_2 & \mathit{write\_in\_cache}(C_2, a, w, v) = C_3 \end{array}}{\texttt{store}(a, v) \vdash_{id} (C, M) \xrightarrow[\mathbf{H}]{\alpha \cdot \beta, \bot} (C_3, M')}$$

The rule considers the case where the address a is not cached. Suppose that, according to the eviction policy, the way to evict w contains the content of the address a'. If the cache line is dirty, $\mathit{write\_back}(C_1, a', w)$ writes in memory the content of the cache line and generates the trace $\alpha = \mathbf{WB}(\mathit{tag}\&\mathit{set}(a'))$. The content of the address a is fetched from memory and generates the trace $\beta = \mathbf{F}(\mathit{tag}\&\mathit{set}(a))$. Eventually, the value v is written in the way w of the cache set of the address a.

Guideline for Secure Locks Using this interface, we have implemented several policies: *write-through* (data is always written in cache and in memory) with *write-allocate* (on store cache-miss, first allocate line in cache) or *write-around* (on store cache-miss, only write in memory), and *write-back* (only write in cache, write back when evicting lines from the cache). The implementation of all these variants of cache policies can be found in the Coq development.

An important secure guideline is that the *state* of a locked cache line must not change when it is accessed for a `load` or a `store`. It follows that the eviction policy must be independent of the previous accesses to locked cache lines. This can be enforced by keeping the usage information of a cache set unmodified for every memory request accessing a locked address. Another subtle point is the handling of the `unlock` request. After the lock is released, when the cache line is eventually evicted, it is necessary to write the cache line to memory even if the cache line has not been modified. Otherwise, by observing a $\mathbf{WB}(t, s)$ event at eviction time, an attacker would deduce that the cache line was modified under lock. To prevent this situation, several options are secure. In our implementation, the `lock` request always sets the dirty bit and the `unlock` request resets the lock bit. Eventually, at eviction time, the cache line will be written back regardless of its previous use. Another option is to eagerly write the cache line to memory at the time of the `unlock` request.

Example 2. Consider again the code of Ex. 1. For a memory access at an address $\texttt{sec} + i$ that is not locked, we may get the events: i) **Hit** if the value is cached;

Algorithm	AES(bs)	AES(l)	Blowfish	DES	RC4	CAST
Lock size	✗	1320 B	4 kB	512 B	256 B	4 kB
Overhead $< 1\%$	✗	1 kB	8 kB	16 B	256 B	4 kB
# Cycles (for 10 kB)	$8.7 \cdot 10^6$	$2.2 \cdot 10^6$	$8.0 \cdot 10^5$	$1.0 \cdot 10^8$	$8.2 \cdot 10^5$	$2.0 \cdot 10^6$

Table 1. Block Ciphers: lock size, overhead and running time

ii) $\mathbf{F}(tag\&set(\texttt{sec} + i))$ if the value is read from RAM; iii) $\mathbf{WB}(tag\&set(\texttt{sec} + i)$ to synchronize the cache with the RAM; iv) but also some other unrelated $\mathbf{WB}(t, s)$ to evict cache lines or some other $\mathbf{F}(t, s)$ modeling prefetching. For a memory access at a locked address $\texttt{perm} + \texttt{sec}[i]$, the hardware guarantees a **Hit** event. Moreover, the `unlock` request ensures that there will always be a $\mathbf{WB}(tag\&set(\texttt{perm} + i))$ event.

4 Evaluation

To evaluate the overhead induced by our locking mechanism to get a Constant-Time Secure (CTS) program, we consider standard blocks ciphers, algorithms from the GhostRider benchmark [20] and an original sorting algorithm based on Batcher's sort [9]. As a baseline, we take non-CTS reference implementations and, when possible, state-of-the-art CTS implementations using other CT programming techniques *e.g.*, bitslicing. We simulate an in-order mono-thread 32-bit RISC-V (RV32I) processor with a L1d cache. The cache contains 128 cache sets, each consisting of 4 ways, each holding 16-byte cache lines.

The block ciphers we consider use lookup tables (*e.g.*, S-boxes) that are vulnerable to timing attacks. With our locking mechanism, it suffices to lock the table to get a CTS program. As the number of table accesses is proportional to the size of the plain text, the overhead cause by locking the lookup table is amortised over time. In Table 1, we show for each block cipher the amount of data needed so that the encryption overhead gets below 1%. We also give the amount of locked data and the number of cycles taken to encrypt 10kB of plain text. AES(bs)[4] is a secure bitsliced implementation of AES. AES(l) and all other implementations are obtained by locking the S-boxes. Our results show that AES(l) is more efficient than the highly optimised AES(bs) implementation. In general, the overhead is amortised for fairly small amount of data (from 256 B to 8 kB). Moreover, the amount of locked data is also relatively small and only occupy at most 50% of the whole cache.

We also evaluate our cache locking mechanism over the programs from the GhostRider benchmark [20] which are also considered by Miao *et al.* for the evaluation of their own hardware support for constant-time programming [22]: histogram, binary search, permutation, dijkstra, heappop. Unlike block ciphers, the amount of locked data is linear in the size of the problem. As a result, we are limited to instances of the problem which fit within the physical constraints

[4] `https://www.bearssl.org/`

of the cache. Fig. 3 details our result for binary search. The results for other algorithms can be found in Appendix B.

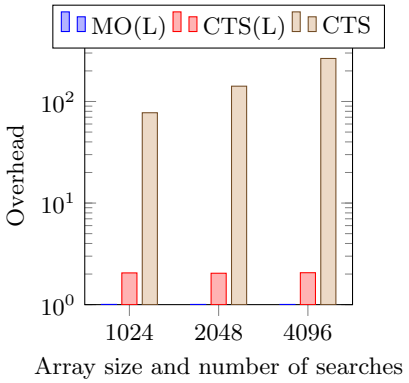

Fig. 3. Binary Search Overhead

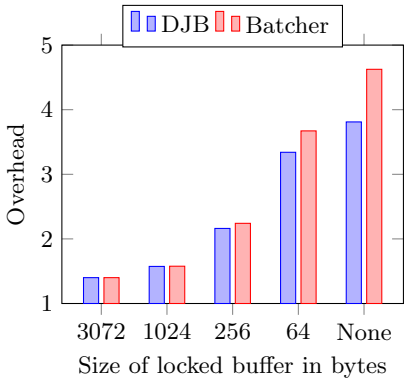

Fig. 4. Overhead of CTS sorts algorithms compared to vulnerable merge-sort.

The Memory Oblivious (MO(L)) binary search is obtained from the textbook baseline binary search by locking the array within the cache before performing the search. The CTS(L) binary search is a CTS binary-search which, in addition to locking the array, is rewritten to avoid secret dependent control-flow. This includes forbidding early return from the main loop and performing branchless tests. The CTS binary search algorithm is actually a linear traversal of the array because, without hardware support, there is no other way to get a constant-time search. To amortise the cost of locking, we perform n successive searches where n is the array size. For the MO(L) and CTS(L) versions, as expected, the overhead remain mostly constant as the algorithms still have the $\mathcal{O}(log(n))$ complexity. Given that locking gets amortised, the MO(L) version incurs very little overhead compared to the CTS(L) where the body of the search code is modified. The linear CTS version, without hardware locking, is not competitive. This highlights the benefit of our locking mechanism.

For our previous experiments, the size of the locked data is determined by the algorithms. Therefore, we have the physical constraint that the locked data fits within the cache. Here, we present a novel CTS sorting method where the size of the locked data K is a parameter which can be tuned to improve efficiency. This novel method is a hybridization of a CTS merge-sort and any network sorting algorithms. Network sorts consist in swapping elements using a pre-determined sequence of pairs of indices which only depends on the size n of the array. Hence, they are CTS by construction.

In our case, instead of swapping single elements, we propose to swap chunks of sorted arrays of size $K/2$. Given an array A of size N and a parameter K, we lock two arrays L and H of size $K/2$. Using the array L as a scratchpad,

each chunk of A of size $K/2$ is sorted using a CTS merge-sort. Then, for each pair (i, j) of the sorting network, we copy the chunk i of the array A in the locked array L and the chunk j of the array A in the locked array H. We then perform the merging of the two sorted array and write the $K/2$ smallest values in the chunk i and the $K/2$ biggest values in the chunk j. Because the arrays L and H are locked, it is feasible to have a CTS merging procedure that runs in $\mathcal{O}(K)$. We applied this hybridization on two different networks sorts : Batcher's sort [9], and Daniel J. Bernstein's portable sort (DJBsort)[4].

CTS sorting algorithms are necessary for implementing the Post-Quantum NIST candidates NTRU and Classic McEliece which sort arrays of 768 32-bit integers[4]. In Fig. 4, we show the overhead of CTS sorting algorithms w.r.t a vulnerable merge-sort for sorting an array of 768 32-bit integers. We consider our hybrid sort with a locked buffer of size K ranging from $64B$ to $3kB$ (i.e. a buffer of the same size as the array of 768 integers, which results in a pure CTS merge-sort without the use of the network). We also compared the hybrids with the original version of both DJB's portable sort and Batcher's sort (i.e. with no locked buffer). The results show that the hybrids outperform both network sorts with any K greater or equal to 64 bytes, and that the overhead decreases as the locked size increases.

Take-away Our locking mechanism has the advantage that, once the relevant tables are locked, we get memory oblivious programs [21] running as fast as the vulnerable code. For block ciphers, as the locking cost gets amortised, we get CTS implementations running as fast as the vulnerable code. Moreover, for our RISC-V microcontroller, our CTS AES with locking outperforms an optimised bitsliced implementation with the additional advantage of keeping the reference implementation mostly unchanged. When the lock size is linear in the problem, we are limited to problem sizes fitting the physical limitations of the cache. Moreover, multiple runs of the algorithm are needed to amortise the locking cost. Yet, the locking mechanism significantly improves scalability compared to standard constant-time programming techniques. Our CTS sorting algorithm also shows that the amount of locked data can be used as a parameter to tune efficiency.

5 Observational Non-Interference with Attacker

In subsection 5.1, we introduce a model where a defender $\mathcal{D}$-program is scheduled with an attacker $\mathcal{A}$-program. In subsection 5.2, we then provide a reasoning principle which isolates the proof obligations pertaining to the $\mathcal{D}$-program and the $\mathcal{A}$-program. Eventually, we instantiate the proof principle for our setting where the $\mathcal{D}$-program and the $\mathcal{A}$-program share a memory cache.

[4] `https://sorting.cr.yp.to`

5.1 Semantics of Instructions and Processes

We consider a set of instructions representative of the RISC-V ISA with arithmetic, jump and memory instructions, where rs, rd represents registers and op contains the operation to apply and a potential immediate value.

$$\begin{aligned} instr ::= & \texttt{arith}(rd, rs_1, rs_2, op) \mid \texttt{jal}(rd, rs_1, rs_2, op) \\ & \mid \texttt{load}(rd, rs_1) \mid \texttt{store}(rs_1, rs_2) \mid \texttt{lock}(rs_1) \mid \texttt{unlock}(rs_1) \end{aligned}$$

The semantics of instructions is parameterised by the memory sub-system. In our model, the code is stored in a separate read-only memory C. A transition takes the form $(R, M) \rightarrow^{\tau}_{id} (R', M')$ where R, R' model the state of the architectural registers of process id, M, M' represent the memory state (including the cache) and τ is the leakage trace induced by the transition. Each transition consists in fetching the instruction i stored at the address $R(PC)$ in the code segment C and updating the registers and memory according to i. The leakage τ of an instruction consists in the leakage due to the memory requests (if any) and the leakage due to the instruction itself. For our processor, the ALU instructions (except division) take a single cycle. Therefore, for $op \neq \texttt{div}$, we have $\mathcal{L}^{\texttt{arith}}(op, v_1, v_2) = \bullet$. The number of cycles of a division depends on the number of bits of the denominator. Therefore, we have $\mathcal{L}^{\texttt{arith}}(div, v_1, v_2) = log_2(v_2)$. Our jump instruction $\texttt{jal}(rd, rs_1, rs_2, op)$ is a *jump and link* which sets rd to the next instruction and sets the program counter according to $op(rs_1, rs_2)$. A regular jump is a `jal` where rd is the immutable zero-register. In term of leakage, we make the worst-case assumption and leak the destination of jumps. As a result, if a jump sets the program counter to the value v, the transition leaks the value v. The details of the semantic rules are given in Appendix C.

Without loss of generality, we consider two processes $id \in \{\mathcal{D}, \mathcal{A}\}$ where $\mathcal{D}$ stands for *defender* and $\mathcal{A}$ stands for *attacker*. Both processes are scheduled according to a deterministic policy and share a memory and a cache. In the model, each process has its own private copy of registers. In a concrete implementation, at context-switch time, the operating system would save and restore registers at fixed predefined known memory locations. As a result, the state of the system is of the form $(r_{\mathcal{D}}, r_{\mathcal{A}}, m, s)$ where $r_{id} \in \texttt{REGISTERS} \rightarrow \texttt{VALUE}$, m is a memory equipped with a cache and s is the scheduler state. At each transition, depending on the scheduler policy, one of the process performs a transition and the state of the scheduler is updated by the leakage trace of the instruction. This models instruction-based but also scheduling strategies based on memory access patterns. This excludes malicious schedulers which inspecting the memory content. With the caveat of Subsection 5.4, this also allows time-based schedulers.

5.2 ONI Preservation Principle with Attacker

Our proof principle differs from Barthe *et al.* [8] because, unlike secure compilation, we consider an explicit attacker that may be scheduled with the secure program. A technical consequence is that we rely on a notion of backward simulation (not a forward simulation). This is needed to cope with situations

where the attacker may perform a denial of service attack and prevent the secure program from executing by monopolising the shared cache. We adapt their lock-step principle [8, Theorem 1] so that it uses a backward simulation (see Definition 3) instead of a forward simulation and also extend the notion of lock-step 2-simulation [8, Def. 6] with a preservation property of final states (see Definition 4).

Definition 3 (Backward simulation). *A relation $\approx$ is a backward lockstep-simulation between a hardware program model $(I_{\mathbf{H}}, \cdot \underset{\mathbf{H}}{\longrightarrow} \cdot)$ and a software program models $(I_{\mathbf{S}}, \cdot \underset{\mathbf{S}}{\rightarrow} \cdot)$ iff:*

- *for every hardware step $\alpha \underset{\mathbf{H}}{\longrightarrow} \beta$, and every software state a such that $a \approx \alpha$, there exists a software state b and a software step $a \underset{\mathbf{S}}{\rightarrow} b$ such that $b \approx \beta$.*
- *the initial states are in relation $\forall i \in \mathcal{I}, \alpha \in I_{\mathbf{H}}(i).\, \exists a \in I_{\mathbf{S}}(i).\, a \approx \alpha$*

Definition 4 (Lockstep 2-simulation). *Given equivalence relations ϕ and ψ over the initial states, ($\equiv_{\mathbf{S}}$,$\equiv_{\mathbf{H}}$) is a lockstep 2-simulation with respect to $\approx$ betweem program models $(I_{\mathbf{S}}, \cdot \underset{\mathbf{S}}{\rightarrow} \cdot)$ and $(I_{\mathbf{H}}, \cdot \underset{\mathbf{H}}{\longrightarrow} \cdot)$ iff*

- *For all software steps $a \overset{t}{\underset{\mathbf{S}}{\rightarrow}} b$ and $a' \overset{t}{\underset{\mathbf{S}}{\rightarrow}} b'$ such that $a \equiv_{\mathbf{S}} a'$ and for all hardware steps $\alpha \overset{\tau}{\underset{\mathbf{H}}{\rightarrow}} \beta$ and $\alpha' \overset{\tau'}{\underset{\mathbf{H}}{\rightarrow}} \beta'$ such that $\alpha \equiv_{\mathbf{H}} \alpha'$, if the states are in the simulation relation $a \approx \alpha$, $a' \approx \alpha'$, $b \approx \beta$ and $b' \approx \beta'$, we have $b \equiv_{\mathbf{S}} b'$, $\beta \equiv_{\mathbf{H}} \beta'$ and $\tau = \tau'$.*
- *Given (i, i') such that $\phi(i, i')$, initial software states $(a, a') \in I_{\mathbf{S}}(i) \times I_{\mathbf{S}}(i')$, we have $a \equiv_{\mathbf{S}} a'$.*
- *Given (i, i') such that $\psi(i, i')$, initial hardware states $(\alpha, \alpha') \in I_{\mathbf{H}}(i) \times I_{\mathbf{H}}(i')$, we have $\alpha \equiv_{\mathbf{H}} \alpha'$.*
- *For all software steps $a \overset{t}{\underset{\mathbf{S}}{\rightarrow}} b$ and $a' \overset{t}{\underset{\mathbf{S}}{\rightarrow}} b'$ such that $a \approx \alpha$, $a' \approx \alpha'$, $a \equiv_{\mathbf{S}} a'$ and $\alpha \equiv_{\mathbf{H}} \alpha'$, we have $\alpha \in \mathcal{F} \iff \alpha' \in \mathcal{F}$.*

However, the Preservation of constant-time policy Theorem (see [8, Theorem 1]) is inadequate in our context because it does not account for an attacker. We propose a more general proof principle. Intuitively, we require that the steps performed by the attacker generate the same traces and preserve the relations $\approx$ and $\equiv$ according to Definition 5.

Definition 5 (Attacker Simulation). *A pair of relations $(\approx, \equiv)$ form an attacker simulation iff*

- *Given $\alpha \overset{\tau}{\underset{\mathcal{A}}{\rightarrow}} \beta$ such that $a \approx \alpha$, we have $a \approx \beta$.*
- *Given $\alpha \overset{\tau}{\underset{\mathcal{A}}{\rightarrow}} \beta$ such that $\alpha \equiv \alpha'$, there exists β' such that $\alpha' \overset{\tau}{\underset{\mathcal{A}}{\rightarrow}} \beta'$ and $\beta \equiv \beta'$.*

Our ONI proof principle is given by Theorem 1.

Theorem 1 (ONI preservation with attacker). *Consider a software execution* $S = (I_{\mathbf{S}}, \cdot \xrightarrow[\mathbf{S}]{t} \cdot)$ *and a hardware execution* $H = (I_{\mathbf{H}}, \cdot \xrightarrow[\mathcal{D}]{t} \cdot \cup \cdot \xrightarrow[\mathcal{A}]{t} \cdot)$ *built from a defender* $D = (I_{\mathbf{H}}, \cdot \xrightarrow[\mathcal{D}]{t} \cdot)$ *and an attacker* $A = (I_{\mathbf{H}}, \cdot \xrightarrow[\mathcal{A}]{t} \cdot)$. *If we have:*

- *A lock-step backward simulation* $\approx$ *between* S *and* D*;*
- *A lock-step 2-simulation* $(\equiv_{\mathbf{S}}, \equiv_{\mathbf{H}})$ *w.r.t* $\approx$ *between* S *and* D*;*
- $(\approx, \equiv_{\mathbf{H}})$ *is an attacker simulation for* A*;*
- *For equivalent hardware states* $\alpha \equiv_{\mathbf{H}} \alpha'$*, the transitions of the attacker and defender are mutually exclusive i.e., it is impossible to have two transitions* $\alpha \xrightarrow[\mathcal{D}]{\tau} \beta$ *and* $\alpha' \xrightarrow[\mathcal{A}]{\tau'} \beta'$ *for some* τ, τ', β, β'*;*

then if S *is observationally non-interferent then* H *is also observationally non-interferent i.e.*

$$ONI(\phi, S) \implies ONI(\psi, H)$$

The proof can be found in Appendix A. In the following, we show how to instantiate Theorem 1 to our model where a $\mathcal{D}$-program and an $\mathcal{A}$-program share a memory cache.

5.3 Simulation and Indistinguishability

To apply Theorem 1, we define a simulation relation $\approx$ as well as software and hardware equivalence relations $\equiv_{\mathbf{S}}$ and $\equiv_{\mathbf{H}}$. Given a software state $S = (r_{\mathcal{D}}, (L, m))$ and a target state $T = (r_{\mathcal{D}}, r_{\mathcal{A}}, (C, M), s)$, the simulation relation holds *i.e.*, $S \approx T$ if *(a)* the defender registers are the same; *(b)* for defender addresses, reading from source memory m is identical to reading from the target memory interface (C, M); and *(c)* the set of locked addresses in L are also locked in the concrete cache C. Additionally, the simulation relation states invariants of the target memory interface, i.e. that the content of non-dirty cache lines is synchronized with the memory and that locked lines are always dirty.

Then, we define equivalence relations at the software ($\equiv_{\mathbf{S}}$) and hardware levels ($\equiv_{\mathbf{H}}$). Two equivalent states will be indistinguishable from the point of view of an attacker. Because we consider that the attacker knows the defender code, two memory states are equivalent if their content is the same for code addresses and for attacker addresses (but their content can be different for addresses in the defender's address space). Two software memory sub-systems (L_1, m_1) and (L_2, m_2) are equivalent if the sets of locked lines for each cache set are the same in L_1 and L_2 and if memories m_1 and m_2 are equivalent. We lift this equivalence to software states $(R_1, (L_1, m_1))$ and $(R_2, (L_2, m_2))$ by requiring that $R_1(PC) = R_2(PC)$ and the memory sub-systems are equivalent. Indeed, we consider that the attacker can deduce the program counter of the victim at any point of its execution, since the branch instruction leak their destination.

Two hardware caches C_1 and C_2 are equivalent if for every set s, the eviction policy usage is the same in both caches, and all cache lines in the set are equivalent, i.e. their validity, dirty and locked bits are equal, their tags are equal,

and when the line contains code or attacker data, the contents are equal (i.e., contents can only differ if it belongs to defender data). Two hardware memory sub-systems (C_1, M_1) and (C_2, M_2) are equivalent if C_1 and C_2 are equivalent hardware caches and m_1 and m_2 are equivalent memories. We lift this equivalence to hardware states $(R^1_{\mathcal{D}}, R^1_{\mathcal{A}}, (C_1, M_1), S_1)$ and $(R^2_{\mathcal{D}}, R^2_{\mathcal{A}}, (C_2, M_2), S_2)$ by requiring that $R^1_{\mathcal{D}}(PC) = R^2_{\mathcal{D}}(PC)$, $R^1_{\mathcal{A}} = R^2_{\mathcal{A}}$, the memory sub-systems are equivalent, and the scheduler states S_1 and S_2 are the same.

Our proof of ONI preservation uses Theorem 1 with respect to a generic memory interface, which we instantiate with three cache policies: write-through with write-allocate or with write-around, and write-back. Among the properties we require on the implementation of the target cache, we have the following property, necessary to prove the first part of Definition 4.

$$\rho_1 \vdash_{\mathcal{D}} \alpha_1 \xrightarrow[\mathbf{H}]{\tau_1, v_1} \beta_1 \wedge \rho_2 \vdash_{\mathcal{D}} \alpha_2 \xrightarrow[\mathbf{H}]{\tau_2, v_2} \beta_2 \wedge \rho_1 \equiv \rho_2 \wedge \alpha_1 \equiv_{\mathbf{H}} \alpha_2 \Rightarrow$$
$$(\mathit{locked}(\alpha_1, \rho_1) \wedge \mathit{locked}(\alpha_2, \rho_2) \wedge \mathit{is\_load\_store}(\rho_1)) \vee \mathit{tag\&set}(\rho_1) = \mathit{tag\&set}(\rho_2)$$
$$\Rightarrow \beta_1 \equiv_{\mathbf{H}} \beta_2 \wedge \tau_1 = \tau_2$$

This says that for any two memory requests ρ_1 and ρ_2 that are equivalent (i.e. they are both the same kind of request) starting in two equivalent states α_1 and α_2, the resulting states β_1 and β_2 will be equivalent and the traces generated by the memory requests will be the same, if:

- either the requests are loads or stores, and both addresses are locked (hence not necessarily the same addresses); or
- the addresses of both requests are in the same tag and set.

These conditions closely match the leakage of memory operations at the software level: loads and stores leak nothing if accessing locked addresses; in other circumstances the tag and set of accessed addresses is leaked. The hardware traces τ_1 and τ_2 will be equal, but their shape might differ from that of software traces: a software event corresponding to an unlocked load may become, in the hardware trace, a simple **Hit** event, or write-back and fetch events $\mathbf{WB}(\cdot)\cdot\mathbf{F}(\cdot)$, depending on whether the corresponding cache line is already in cache, or whether a cache line has to first be evicted, written back to memory, before the cache line of interest is fetched from the memory. The complete set of properties required of a hardware memory interface can be found in the Coq development.

5.4 Discussion

Absence of remaining timing leakage. Our security theorem (see Theorem 1) establishes that hardware leakage traces are indistinguishable. Yet, our model is still too abstract to claim the complete absence of timing leaks *i.e.*, a form of cycle accurate constant-time. Here, we give some insights why the claim may hold for our RISC-V micro-controller. More precisely, we argue (informally) that the leakage trace provides the necessary information to reconstruct the state of the micro-architecture and provide a cycle-accurate simulation. Our hardware

leakage precisely models the cache and memory interaction and, therefore, the state of the memory cache can be reconstructed. The instruction cache is not precisely modelled. Yet, as the control decisions are leaked, the current instruction is known as well as the content of the instruction cache. The remaining component which has a timing influence is the pipeline which may stall. However, our pipeline state only depends on the sequence of instructions and may only stall if results are not ready due to memory operations or multi-cycle arithmetic operations. Memory operations are precisely modelled in the leakage trace. Moreover, the only multi-cycle arithmetic operation is division (it depends on the number of bits of the dividend which is leaked in the trace). Though a formal statement would require a precise hardware model, we are confident that our formal theorem ensures the absence of timing leakage at the cycle level.

Handling of lock interrupts In the model, the execution gets stuck if there is no available cache line for locking an address. In practice, this raises a hardware interrupt. We describe various ways to handle this situation using some OS support. A first solution is to ensure statically that this situation never occurs using a resource allocation *e.g.* the Banker's algorithm [13]. For an IoT setting with a static number of processes, this looks like a feasible solution. Another solution is to pause the process and re-schedule it when the address can be locked. This requires that another process unlocks a cache line in the same cache set. This approach would fit nicely in our formal model but has the drawback that deadlocks may occur. A last solution would be to preemptively unlock a memory address from another process and relock it during the context-switch. As this requires some bookkepping on the OS side, this mechanism does not directly fit our formal model where the OS is abstracted away.

6 Related Work

Countermeasures against cache attacks exist at both the software and hardware level. At software level, the cryptographic constant-time (CT) discipline which combines the program counter model [23] with memory obliviousness [21] is the *de facto* standard [10]. Our software model relaxes CT so that locked cache lines are indistinguishable. Barthe *et al.* [5, 3, 4] study a similar relaxation using the concept of stealth memory [19] for a sophisticated system including a MMU and hypercalls. Our model is significantly simpler while modelling a concrete hardware implementation of a stealth memory. Our locking mechanism is also directly available using an ISA extension whereas, in the model of Barthe *et al.*, stealth memory is managed at the OS level. Guarnieri *et al.* [18] provide a hardware/software contract between an architectural and a hardware semantics with speculation. For a more sophisticated architecture, their guarantee is similar to ours. However, they do not consider a locking mechanism. In the context of secure compilation, Barthe *et al.* propose a general framework for preserving ONI [8] which has been successfully applied to prove correct a CT-preserving CompCert compiler [6]. We generalise this theory and propose a proof principle which clearly separates the proof ogligation of a defender and an attacker.

Writing CT programs is notoriously hard. FaCT [12] is a DSL performing program transformations to turn a well-typed program into a CT program. FaCT could benefit from our cache locking mechanism in order to deal with secret-dependent memory accesses and therefore accept more programs. More generally, we are confident that static verification tools for verifying CT programs *e.g.,* [15, 2, 11], could be adapted with little modification to capture the behaviour of locked memory accesses.

At hardware level, there are several approaches to protect against cache attacks. NoMo Cache [14] partitions the cache ways on a process basis. This has the advantage of improving the security in a transparent manner. Yet, there is some information leakage if the data does not fit in the private cache set. Our locking mechanism is more fined-grained but it is the responsibility of the software to lock sensitive accesses. Miao and al. [22] propose a cache design where sensitive data may be evicted. This augments the availability of the cache but requires more hardware resources and complex macro instructions in order to securely reload the evicted cache lines. Our solution is more lightweight and secure accesses are always cached at the cost of reserving resources through the locking mechanism. PLcache [27] proposes a locking mechanism which improves the security but may leak information in rare cases. We study variations of the design of Gaudin *et al.* [16] which fixes the potential information leakage of PLcache. Oblivous RAM [25] (ORAM) randomises the memory access so that an attacker may only observe noise. The mechanism requires more hardware resources, in particular a strong Random Number Generator, than our locking mechanism. The security proof would also need a notion of probabilistic ONI that is out of reach of our reasoning principle.

7 Conclusion

We provide a hardware/software contract for a cache locking mechanism expressed as the preservation of Observational Non-Interference. Our proof principles separate the proof-obligations that are required on the defender and attacker. We also identify the proof-obligations that are required for the cache implementation and provide mechanised proofs for common caching strategies. We provide a simple software leakage model where locked accesses are not visible to an attacker and show the property that need to be enforced by the hardware implementation to enforce security. We also leverage the locking mechanism to secure existing algorithms at little engineering cost. We also show how to exploit locking to speed-up constant-time sorting. As future work, it would be interesting to investigate whether the locking mechanism could be adapted to the instruction cache. This would allow to further relax the CT model and implement countermeasures which balance conditionals [1, 7] while mitigating cache attacks. Another interesting direction is to push the verification effort further and verify formally that the actual hardware implementation complies with our hardware contract.

References

[1] J. Agat. "Transforming Out Timing Leaks". In: *POPL*. ACM, 2000.

[2] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. "Verifying Constant-Time Implementations". In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 53–70.

[3] G. Barthe, G. Betarte, J. D. Campo, and C. Luna. "System-Level Non-interference of Constant-Time Cryptography. Part I: Model". In: *J. Autom. Reason.* 63.1 (2019).

[4] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie. "System-Level Non-interference of Constant-Time Cryptography. Part II: Verified Static Analysis and Stealth Memory". In: *J. Autom. Reason.* 64.8 (2020).

[5] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie. "System-level Non-interference for Constant-time Cryptography". In: *CCS*. ACM, 2014, pp. 1267–1279.

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

[7] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie. "Secure Compilation of Constant-Resource Programs". In: *CSF*. IEEE, 2021, pp. 1–12.

[8] G. Barthe, B. Grégoire, and V. Laporte. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"". In: *CSF*. IEEE Computer Society, 2018, pp. 328–343.

[9] K. E. Batcher. "Sorting Networks and Their Applications". In: vol. 32. AFIPS Conference Proceedings. Thomson Book Company, 1968.

[10] D. J. Bernstein, T. Lange, and P. Schwabe. "The Security Impact of a New Cryptographic Library". In: *LATINCRYPT*. Vol. 7533. LNCS. Springer, 2012, pp. 159–176.

[11] S. Blazy, D. Pichardie, and A. Trieu. "Verifying constant-time implementations by abstract interpretation". In: *J. Comput. Secur.* 27.1 (2019), pp. 137–163.

[12] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. "FaCT: a DSL for timing-sensitive computation". In: *PLDI*. ACM, 2019, pp. 174–189.

[13] E. W. Dijkstra. "The mathematics behind the banker's algorithm". In: Berlin, Heidelberg: Springer-Verlag, 1982. ISBN: 0387906525.

[14] L. Domnitser, A. Jaleel, J. Loew, N. B. Abu-Ghazaleh, and D. Ponomarev. "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks". In: *ACM Trans. Archit. Code Optim.* 8.4 (2012), 35:1–35:21.

[15] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. "CacheAudit: A Tool for the Static Analysis of Cache Side Channels". In: *ACM Trans. Inf. Syst. Secur.* 18.1 (2015), 4:1–4:32.

[16] N. Gaudin, J. Hatchikian-Houdot, F. Besson, P. Cotret, G. Gogniat, G. Hiet, V. Lapotre, and P. Wilke. "Work in Progress: Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections". In: *EuroS&P Workshops*. IEEE, 2023, pp. 304–310.

[17] Q. Ge, Y. Yarom, and G. Heiser. “No Security Without Time Protection: We Need a New Hardware-Software Contract”. In: *APSys*. ACM, 2018, 1:1–1:9.
[18] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. “Hardware-Software Contracts for Secure Speculation”. In: *SP*. IEEE, 2021, pp. 1868–1883.
[19] T. Kim, M. Peinado, and G. Mainar-Ruiz. “STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud”. In: *USENIX Security Symposium*. USENIX Association, 2012, pp. 189–204.
[20] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi. “GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation”. In: *ASPLOS*. ACM, 2015, pp. 87–101.
[21] C. Liu, M. Hicks, and E. Shi. “Memory Trace Oblivious Program Execution”. In: *CSF*. IEEE Computer Society, 2013, pp. 51–65.
[22] Y. Miao, M. T. Kandemir, D. Zhang, Y. Zhang, G. Tan, and D. Wu. “Hardware Support for Constant-Time Programming”. In: *MICRO*. ACM, 2023, pp. 856–870.
[23] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner. “The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks”. In: *ICISC*. Vol. 3935. LNCS. Springer, 2005.
[24] M. Mushtaq, M. A. Mukhtar, V. Lapotre, M. K. Bhatti, and G. Gogniat. “Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA”. In: *Inf. Syst.* 92 (2020), p. 101524.
[25] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *J. ACM* 65.4 (2018), 18:1–18:26.
[26] E. Tromer, D. A. Osvik, and A. Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. In: *J. Cryptol.* 23.1 (2010), pp. 37–71.
[27] Z. Wang and R. B. Lee. “New cache designs for thwarting software cache-based side channel attacks”. In: *ISCA*. ACM, 2007, pp. 494–505.
[28] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. USENIX Association, 2014, pp. 719–732.

A Proof of Theorem 1

Proof. To prove $ONI(\psi, T)$, we suppose that we have two derivations $\alpha \xrightarrow[\mathbf{H}]{t}{}^{n} \beta$ and $\alpha' \xrightarrow[\mathbf{H}]{t'}{}^{n} \beta'$ and prove that the traces are the same $t = t'$ and that $\beta \in \mathcal{F}(T) \iff \beta' \in \mathcal{F}(T)$. By definition of ONI, there is a pair of input (i, i') such that $\phi(i, i')$ and $\alpha \in I_{\mathbf{H}}(i)$ and $\beta \in I_{\mathbf{H}}(i')$. There is also a pair of software states $(a, a') \in I_{\mathbf{S}}(i) \times I_{\mathbf{S}}(i')$. Because the initial states are in relation by backward simulation, we have $a \approx \alpha$ and $a' \approx \alpha'$. By definition of the lockstep 2-simulation we also have $a \equiv_{\mathbf{S}} a'$ and $\alpha \equiv_{\mathbf{H}} \alpha'$.

The proof is by induction over the length of the derivation.

– Base case. A 0-step derivation generates the empty trace ϵ. As a result, the traces are equal. It remains to prove that $\alpha \in \mathcal{F} \iff \alpha' \in \mathcal{F}$. By definition of *ONI* for a 0-step derivation, we have that $a \in \mathcal{F} \iff a' \in \mathcal{F}$.
  - If $a \in \mathcal{F}$, then $a' \in \mathcal{F}$, and by definition of the backward simulation, we get that $\alpha \in \mathcal{F}$ and $\alpha' \in \mathcal{F}$ and the property holds.
  - Suppose the $a \notin \mathcal{F}$. As a result, there are software derivations $a \xrightarrow[\mathbf{S}]{t} b$ and $a' \xrightarrow[\mathbf{S}]{t'} b'$ for some t, b, t' and b'. By definition of *ONI*, we get that $t = t'$. The property follows by definition of the lockstep 2-simulation.
– Inductive case. Suppose that we have two derivations of length $n+1$ of the form

$$\alpha \xrightarrow[\mathbf{H}]{\tau_1}{}^{1}\ \beta_1 \xrightarrow[\mathbf{H}]{\tau}{}^{n}\ \beta \text{ and } \alpha' \xrightarrow[\mathbf{H}]{\tau'_1}{}^{1}\ \beta'_1 \xrightarrow[\mathbf{H}]{\tau'}{}^{n}\ \beta'$$

We need to prove that $\tau_1 \cdot \tau = \tau'_1 \cdot \tau'$ and $\beta \in \mathcal{F} \iff \beta' \in \mathcal{F}$. By definition of the backward simulation, we can exhibit two software derivations $a \xrightarrow[\mathbf{S}]{t_1}{}^{1}\ b_1$ and $a' \xrightarrow[\mathbf{S}]{t'_1}{}^{1}\ b'_1$ for some t_1, b_1, t'_1 and b'_1 such that $b_1 \approx \beta_1$ and $b'_1 \approx \beta'_1$. By definition of *ONI*, we get that $t_1 = t'_1$. The property follows by induction hypothesis. □

B Evaluation of Algorithms with Input Dependent Locks

The evaluation results for Histogram, Permuation, Heap-pop and Dijkstra can be found in Fig. 5, Fig. 6, Fig. 7 and Fig. 8.

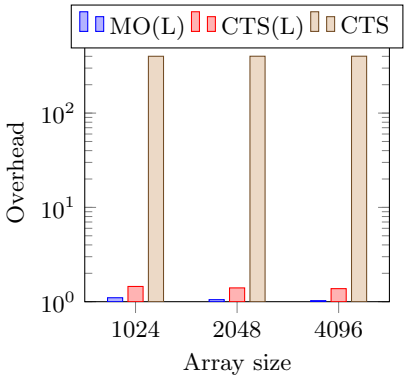

Fig. 5. Histogram Overhead

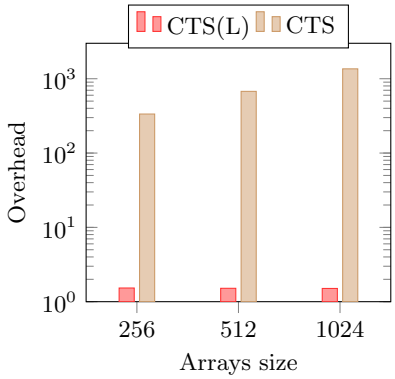

Fig. 6. Permutation Overhead

C Semantics of Instructions

The semantics of instructions is given in Fig. 9.

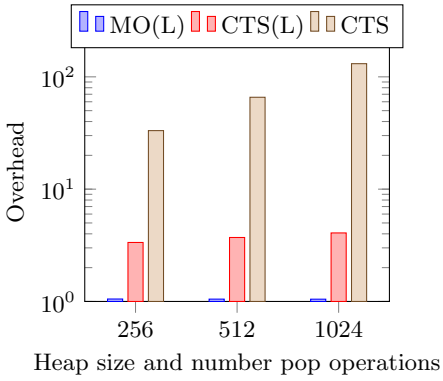

Fig. 7. Heap-pop Overhead

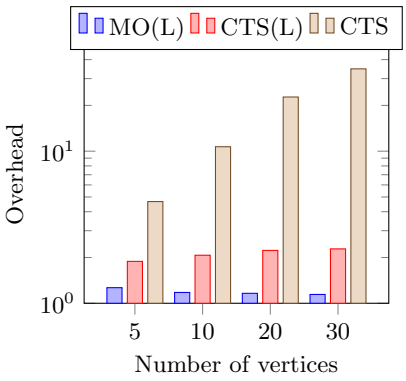

Fig. 8. Dijkstra Overhead

$$\textsc{Decode}\ \frac{R(PC) = \lfloor a \rfloor \qquad \texttt{load}(a) \vdash M \xrightarrow{\alpha,v} M' \qquad decode(v) = \lfloor \texttt{instr} \rfloor}{get\_instr(R, M) \Rightarrow^{t_1} \lfloor \texttt{instr} \rfloor, M'}$$

$$\textsc{Arith}^{\mathbf{P}}\ \frac{\begin{array}{c} get\_instr(R, M) \Rightarrow^{t_1} (\texttt{arith}(rd, rs_1, rs_2, op), M') \\ t_2 = leak_{\texttt{arith}}(op, R(rs_2)) \end{array}}{(R, M) \xrightarrow[\mathbf{P}]{t_1 \cdot t_2} (R[rd \mapsto op(R(rs_1), R(rs_2)), PC \mapsto incr(R(PC))], M')}$$

$$\textsc{Jump}^{\mathbf{P}}\ \frac{\begin{array}{c} get\_instr(R, M) \Rightarrow^{t_1} (\texttt{jump}(rd, rs_1, rs_2, op), M') \\ newPC = op(R(rs_1), R(rs_2), R(PC)) \end{array}}{(R, M) \xrightarrow[\mathbf{P}]{t_1 \cdot newPC} (R[PC \mapsto newPC, rd \mapsto incr(R(PC))], M')}$$

$$\textsc{Load}^{\mathbf{P}}\ \frac{\begin{array}{c} get\_instr(R, M) \Rightarrow^{t_1} (\texttt{load}(rd, rs_1), M') \\ rd \neq PC \qquad \texttt{load}(R(rs_1)) \vdash M' \xrightarrow{t_2,v} M'' \end{array}}{(R, M) \xrightarrow[\mathbf{P}]{t_1 \cdot t_2} (R[rd \mapsto v, PC \mapsto incr(R(PC))], M'')}$$

$$\textsc{Store}^{\mathbf{P}}\ \frac{\begin{array}{c} get\_instr(R, M) \Rightarrow^{t_1} (\texttt{store}(rs_1, rs_2), M') \\ \texttt{store}(R(rs_1), R(rs_2)) \vdash M' \xrightarrow{t_2,\bot} M'' \qquad perm(R(rs_1)) = \texttt{rw} \end{array}}{(R, M) \xrightarrow[\mathbf{P}]{t_1 \cdot t_2} (R[PC \mapsto incr(R(PC))], M'')}$$

$$\textsc{Lock}^{\mathbf{P}}\ \frac{get\_instr(R, M) \Rightarrow^{t_1} (\texttt{lock}(rs_1), M') \qquad \texttt{lock}(R(rs_1)) \vdash M' \xrightarrow{t_2,\bot} M''}{(R, M) \xrightarrow[\mathbf{P}]{t_1 \cdot t_2} (R[PC \mapsto incr(R(PC))], M'')}$$

$$\textsc{Unlock}^{\mathbf{P}}\ \frac{get\_instr(R, M) \Rightarrow^{t_1} (\texttt{unlock}(rs_1), M') \qquad \texttt{unlock}(R(rs_1)) \vdash M' \xrightarrow{t_2,\bot} M''}{(R, M) \xrightarrow[\mathbf{P}]{t_1 \cdot t_2} (R[PC \mapsto incr(R(PC))], M'')}$$

Fig. 9. Program semantics