Directed Automated Memory Performance Testing

Sudipta Chattopadhyay

Singapore University of Technology and Design (SUTD), Singapore
sudipta_chattopadhyay@sutd.edu.sg

Abstract. Understanding software non-functional properties (*e.g.* time, energy and security) requires deep understanding of the execution platform. The design of caches plays a crucial role in impacting software performance (for low latency of caches) and software security (for cache being used as a side channel). We present CATAPULT, a novel test generation framework to systematically explore the cache behaviour of an arbitrary program. Our framework leverages dynamic symbolic execution and satisfiability modulo theory (SMT) solvers for generating test inputs. We show the application of CATAPULT in testing timing-related properties and testing cache side-channel vulnerabilities in several open-source programs, including applications from OpenSSL and Linux GDK libraries.

1 Introduction

Program path captures an artifact of program behaviour in critical software validation process. For instance, in directed automated random testing (in short DART) [15], program paths are systematically explored to attempt path coverage and construct a test-suite for software validation. Several non-functional software properties (*e.g.* performance and security) critically depend on the execution platform and its interaction with the application software. For validating such properties, it is not sufficient to explore merely the program behaviour (*e.g.* program paths), it is crucial to explore *both* program behaviour and its interaction with the underlying hardware components (*e.g.* cache and communication bus). Hence, any technique that systematically explores both the program behaviour and the associated changes in the hardware, can be extremely useful for testing software non-functional properties.

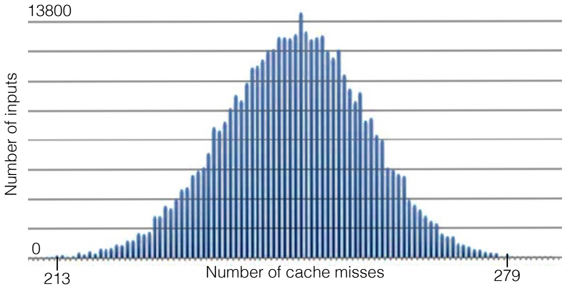


Fig. 1. Distribution of cache misses within a single program path [1]

In order to illustrate our observation, let us consider Figure 1, which specifically records cache performance. We have generated Figure 1 by executing an implementation of Advanced Encryption Standard (AES) [1]. We randomly generated 256000 different inputs to execute a single path of the respective implementation. Figure 1 captures the distribution of the number of inputs w.r.t. the number of observed cache misses [12]. We clearly observe a high variation on cache misses, hence the overall memory performance, even within the scope of a single program path. To solve the problem of systematically exploring cache behaviour and to expose the memory performance of a program, is the main contribution of our paper.

We present CATAPULT – a framework that leverages dynamic symbolic execution and satisfiability modulo theory (SMT) to explore both program behaviour and its associated cache behaviour. CATAPULT takes the binary code and a cache configuration as inputs, and produces a test suite as output. Each test in the test suite exposes a *unique* cache performance (*i.e.* the number of cache misses). Our framework does not generate *false positives*, meaning that the cache performance associated with each test indeed serves as an witness of an execution. Moreover, if our framework terminates, it guarantees to witness all possible cache behaviour in the respective program. Therefore, CATAPULT shares all the guarantees that come with classic approaches based on dynamic symbolic execution [15].

Our approach significantly differs from the techniques based on static cache analysis [20]. Unlike approaches based on static analysis, CATAPULT guarantees the absence of false positives. Moreover, unlike static analysis, CATAPULT generates a witness for each possible cache behaviour. To explore different cache behaviour of a program is, however, extremely involved. This is due to the complex interaction between program artifacts (*e.g.* memory-related instructions) and the design principle of caches. In order to solve this challenge, we have designed a novel symbolic model for the cache. Given a set of inputs, expressed via quantifier-free predicates, such a symbolic model encodes all possible cache behaviour observed for the respective set of inputs. As a result, this model can be integrated easily with the constraints explored and manipulated during dynamic symbolic execution. The size of our symbolic cache model is polynomial with respect to the number of memory-related instructions.

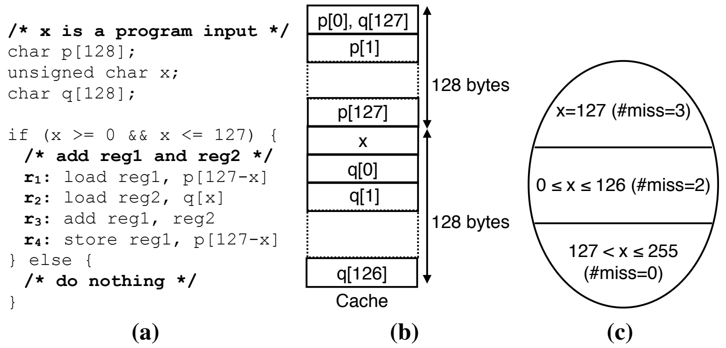
In summary, this paper makes the following contributions:

1. We present a test generator CATAPULT, leveraging on dynamic symbolic execution, to systematically explore the cache behaviour and hence, the memory performance of a program.
2. To show the generality of our approach, we instantiate our framework for two widely used cache replacement strategies – *least recently used* (LRU) and *first in first out* (FIFO).
3. We show the application of CATAPULT in two different contexts – *(i)* for testing timing-related constraints and *(ii)* for testing cache side-channel leakage.
4. We implement our framework on top of a *state-of-the-art* symbolic execution tool KLEE [2] and evaluate it with several cryptographic and device driver routines in OpenSSL library and Linux GDK library. For all the chosen subject programs, exhaustive test input generation is infeasible. However, CATAPULT terminates for all the subject programs and it generates *all tests* within a feasible timing-range from 10 seconds to 4.5 hours. CATAPULT prototype and the obtained results are available for future usage and extension in the following URL:

   https://bitbucket.org/sudiptac/catapult/

2 Background and Overview

Background on caches A cache is a fast memory employed between the CPU and the main memory (DRAM). For a given memory access, the cache is looked up first. A *cache configuration* can be defined by three parameters – cache line size (in bytes), number of cache sets, associativity and replacement policy. In an M-bit memory address, $\mathcal{S}$ bits are reserved to distinguish the cache set in which the respective address is mapped to and $\mathcal{B}$ bits are reserved to distinguish individual bytes within a cache line. For an arbitrary memory address $addr$, we say that it belongs to the *memory block* starting at address $\lfloor \frac{addr}{2^{\mathcal{B}}} \rfloor$. If the content of $addr$ is not found in the cache, $2^{\mathcal{B}}$ consecutive bytes are fetched from the memory address $\lfloor \frac{addr}{2^{\mathcal{B}}} \rfloor$ and they are mapped into the cache set $\lfloor \frac{addr}{2^{\mathcal{B}}} \rfloor \ mod \ 2^{\mathcal{S}}$. Each cache set can only hold as many cache lines as the associativity of the cache. Therefore, if the associativity of the cache is $\mathcal{A}$, the overall size of the cache is $\left(2^{\mathcal{S}} \cdot 2^{\mathcal{B}} \cdot \mathcal{A}\right)$. Finally, since different memory blocks may map into the same cache set, caches store a tag in each cache line to distinguish different memory blocks. Since $(\mathcal{S} + \mathcal{B})$ bits are used to identify cache sets and individual bytes in a cache line, the rest of the bits in the memory address are used as tag. For an $\mathcal{A}$-way set-associative cache, a *cache state* is a set of ordered $\mathcal{A}$-tuples, one for each cache set. Such a tuple captures the set of memory blocks contained in the respective set and the order in which these blocks would be replaced. For example, an ordered pair $\langle m_1, m_2 \rangle$ captures the cache state where m_2 would be replaced before m_1 in a 2-way set-associative cache.



```
/* x is a program input */
char p[128];
unsigned char x;
char q[128];

if (x >= 0 && x <= 127) {
  /* add reg1 and reg2 */
  r1: load reg1, p[127-x]
  r2: load reg2, q[x]
  r3: add reg1, reg2
  r4: store reg1, p[127-x]
} else {
  /* do nothing */
}
```

Fig. 2. (a) a program where cache performance exhibits variation within a program path, (b) mapping of variables in a 256 bytes cache, (c) cache performance with respect to different inputs

Overview In this section, we discuss the motivation behind our approach through the example in Figure 2. For the sake of illustration, we use both assembly-level and source-level syntax in Figure 2(a). However, our test generation is carried out directly on the binary. Let us assume the code shown in Figure 2(a) runs on a platform having direct-mapped (*i.e.* associativity $\mathcal{A} = 1$), 256 bytes cache. The mapping of different variables into the cache is shown in Figure 2(b). We assume that the variable x is also allocated a register in the generated code. Therefore, reading variable x in the code fragment, as shown in Figure 2(a), does not involve any cache access.

Let us assume that we want to check whether the code in Figure 2(a) exhibits *more than two cache misses* when x is a program input. We first execute the program with a random input $x = 0$. We also compute the path condition $x \geq 0 \wedge x \leq 127$ which symbolically encodes all inputs exercising the respective program path. We note that for $x = 0$, both r_1 and r_2 suffer cache misses. For $x = 0$, the store instruction r_4 is a cache hit, as `p[127]` is already loaded into the cache and it was not replaced by `q[0]`.

Since dynamic symbolic execution aims to obtain path coverage, the next test input will be generated by manipulating the path condition and solving the following constraint: $\neg(x \geq 0 \wedge x \leq 127)$. This will result in inputs exercising the `else` branch in Figure 2(a), which, in turn does not access memory.

It is worthwhile to note that classic symbolic execution may not reveal critical inputs related to cache performance. For instance, executing the code in Figure 2(a), for $x = 127$, will access `p[0]`, `q[127]` and `p[0]` in sequence. Since `q[127]` replaces `p[0]` from the cache, all accesses will be cache misses. Figure 2(c) shows the partitioning of the input space according to cache performance.

A classic symbolic-execution-based approach explores program paths instead of cache behaviour. The `if` branch in Figure 2(a) encodes two different cache behaviors – one for inputs $0 \leq x \leq 126$ and another for input $x = 127$. Therefore, it is crucial to devise a methodology that can differentiate inputs based on cache behaviour, even though such inputs exercise the same program path.

How CATAPULT works For each explored program path, `CATAPULT` generates symbolic constraints to encode all possible cache behaviour. For instance, consider the program path captured by the path condition $x \geq 0 \wedge x \leq 127$. Assuming an empty cache, the first load instruction will suffer a cache miss. For instruction r_2, we check whether the memory block containing address `&q[x]` has been accessed for the first time as follows:

$$tag(r_2) \neq tag(r_1) \vee set(r_2) \neq set(r_1)$$

where tag (respectively, set) captures the cache-tag (respectively, cache set) for the memory address accessed by the respective instruction. Intuitively, the aforementioned constraint is true if r_2 accesses a different cache set than r_1 or the memory address accessed by r_2 has a different cache-tag as compared to the memory address accessed by r_1. In such cases r_2 will suffer a cold miss. The constraint is valid as `p` and `q` are different arrays. Similarly, we can check whether r_4 suffers a cold miss as follows:

$$(tag(r_4) \neq tag(r_1) \vee set(r_4) \neq set(r_1)) \wedge (tag(r_4) \neq tag(r_2) \vee set(r_4) \neq set(r_2))$$

This constraint is unsatisfiable, as r_1 and r_4 access the same memory address for all possible inputs. Therefore, r_4 cannot suffer a cold cache miss. To check whether `p[127-x]` can be replaced by r_2 (hence inducing a cache miss at r_4), we use the following set of constraints.

$$(tag(r_2) \neq tag(r_4) \wedge set(r_2) = set(r_4)) \Rightarrow (miss_4 = 1)$$

$$(tag(r_2) = tag(r_4) \vee set(r_2) \neq set(r_4)) \Rightarrow (miss_4 = 0)$$

The variable $miss_4$ indicates whether r_4 is a cache miss or not. `CATAPULT` explores different solutions of $miss_4$. In this example, $miss_4$ is 1 for $x = 127$ and $miss_4$ is 0 for $0 \leq x \leq 126$. Therefore, by systematically generating symbolic constraints and exploring the different solutions, `CATAPULT` can discover that r_4 suffers a cache miss only for input $x = 127$, leading to a total three cache misses in the respective execution.

3 Test Generation

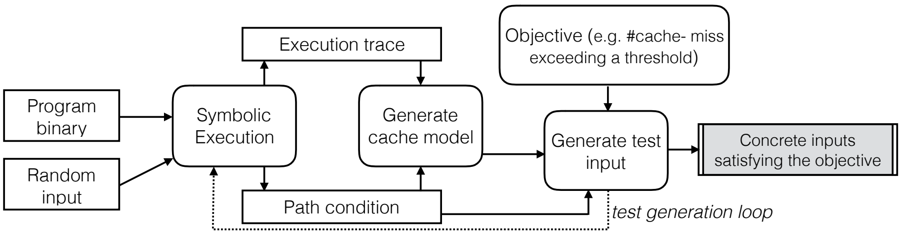


Fig. 3. Our test generation framework

Figure 3 and Algorithm 1 outline all the stages involved in `CATAPULT`. Algorithm 1 takes a program $\mathcal{P}$, the cache configuration $\mathcal{C}$ and an objective $\mathcal{O}$ as inputs. Informally, $\mathcal{O}$ captures dynamic properties related to cache performance. In section 5, we show how $\mathcal{O}$ is formulated to check *(i)* timing-related properties and *(ii)* cache side-channel vulnerabilities. Given the inputs, Algorithm 1 leverages dynamic symbolic execution and computes all unique tests (in the given time budget) that satisfy $\mathcal{O}$.

We first execute $\mathcal{P}$ with a random input $\mathcal{I}$ and compute the path condition Ψ_{path} as well as the execution trace $\mathbb{S}$. The trace $\mathbb{S}$ is captured via a sequence of pairs as follows:

$$\mathbb{S} \equiv \langle (r_1, \sigma_1), (r_2, \sigma_2), \ldots, (r_n, \sigma_n) \rangle \tag{1}$$

Here r_i denotes the i-th memory-related instruction executed and σ_i symbolically captures the memory address accessed by r_i. For example, when we execute the code frag-

Algorithm 1 Test Generation Algorithm

Input: Program $\mathcal{P}$, cache configuration $\mathcal{C}$, objective $\mathcal{O}$.
Output: A test suite $\mathcal{T}$, where each test $t \in \mathcal{T}$ satisfies $\mathcal{O}$

```
$AllPCs$ = $UnchkdPCs$ = $\mathcal{T}$ = empty
Select a random input $\mathcal{I}$
$Explore(\mathcal{P}, \mathcal{C}, \mathcal{I})$
while $UnchkdPCs \neq$ empty do
    select $\varphi \in UnchkdPCs$
    $UnchkdPCs$ := $UnchkdPCs \setminus \{\varphi\}$
    let $\varphi \leftarrow pc_1 \wedge pc_2 \wedge \ldots \wedge pc_{r-1} \wedge pc_r$
    if $\varphi$ is satisfiable then
        $t_\theta \leftarrow$ concrete input satisfying $\varphi$
        $Explore(\mathcal{P}, \mathcal{C}, t_\theta)$
    end if
end while
Report generated test suite $\mathcal{T}$

procedure EXPLORE($\mathcal{P}, \mathcal{C}, t$)
    execute $\mathcal{P}$ on input $t$
    let $\Psi_{path}$ be the path condition
    let $\mathbb{S}$ be the execution trace
    /* Generate the cache model */
    /* See section 4 */
    $\Gamma(\Psi_{path})$ := $CacheModel(\mathcal{C}, \Psi_{path}, \mathbb{S})$
    /* formulate objective (section 5) */
    $\mathcal{O}_{\mathbb{S}}$ := $ObjectivePred(\mathbb{S})$
    /* exclude current solutions */
    /* this step ensures unique tests */
    $\mathcal{O}_{\mathbb{S}}$ := $ExcludeCurTest(\mathcal{T}, \mathcal{O}_{\mathbb{S}})$
    let $\Omega$ := $\Gamma(\Psi_{path}) \wedge \mathcal{O}_{\mathbb{S}} \wedge \Psi_{path}$
    /* Generate relevant tests */
    /* See section 5 */
    while $\Omega$ is satisfiable do
        get $k_\theta$ satisfying $\Omega$
        $\mathcal{T} \bigcup = \{k_\theta\}$
        refine $\mathcal{O}_{\mathbb{S}}$ to exclude solution $k_\theta$
        $\Omega$ := $\Gamma(\Psi_{path}) \wedge \mathcal{O}_{\mathbb{S}} \wedge \Psi_{path}$
    end while
    let $\Psi_{path} \equiv pc_1 \wedge pc_2 \wedge \ldots \wedge pc_u$
    /* build partial path conditions */
    for $i \leftarrow 1, u$ do
        $\varphi_i$ := $pc_1 \wedge pc_2 \wedge \ldots pc_{i-1} \wedge \neg pc_i$
        if $\varphi_i \notin AllPCs$ then
            $AllPCs \bigcup = \{\varphi_i\}$
            $UnchkdPCs \bigcup = \{\varphi_i\}$
        end if
    end for
    /* end exploration of $\Psi_{path}$ */
end procedure
```

ment of Figure 2(a) with input $x = 0$, we obtain the following execution trace:

$$\mathbb{S} \equiv \langle (r_1, \&p + 127 - x), (r_2, \&q + x), (r_3, \&p + 127 - x) \rangle$$

We use the variable $miss_i$ to represent whether r_i is a cache miss (set to 1 if r_i was a cache miss and set to 0 otherwise) for inputs satisfying Ψ_{path}. The value of $miss_i$ depends on all symbolic memory addresses σ_k, where $k \in [0, i)$. Therefore, we bound the value of $miss_i$ through symbolic constraints. In particular, given the execution trace $\mathbb{S}$ and the path condition Ψ_{path}, the procedure *CacheModel* computes $\Gamma(\Psi_{path})$ for cache configuration $\mathcal{C}$ (*cf.* line 21 in Algorithm 1). Such a model $\Gamma(\Psi_{path})$ encodes all possible values of $miss_i$ for all $i \in [1, n]$ and for any input satisfying Ψ_{path}. In section 4, we describe the formulation of $\Gamma(\Psi_{path})$ in detail.

The cache model $\Gamma(\Psi_{path})$ and the path condition Ψ_{path} are used to generate test inputs that satisfy the objective $\mathcal{O}$ (*cf.* lines 31-34). We first extract a predicate $\mathcal{O}_{\mathbb{S}}$ from the execution trace $\mathbb{S}$ that captures such an objective (*cf.* line 23). For example, let us assume our objective is to generate test inputs that suffer at least 1000 cache misses. For an execution trace $\mathbb{S}$, we can simply extract $\mathcal{O}_{\mathbb{S}}$ as $\sum_1^n miss_i \geq 1000$. Subsequently, we can generate a test input that satisfies the following formula:

$$\Gamma(\Psi_{path}) \wedge \left(\sum_{i=1}^{n} miss_i \geq 1000 \right) \wedge \Psi_{path} \tag{2}$$

The refinement of $\mathcal{O}_{\mathbb{S}}$ (line 33) depends on the context. For instance, let us assume that the designer needs to compute (at most) one test for each scenario exhibiting at least 1000 cache misses. In such a case, the following refinement is made to $\mathcal{O}_{\mathbb{S}}$:

$$\mathcal{O}_{\mathbb{S}} = \mathcal{O}_{\mathbb{S}} \wedge \left(\sum_{i=1}^{n} miss_i \neq \sum_{i=1}^{n} miss_i^{(c)} \right)$$

where $miss_i = miss_i^{(c)}$ (for $i \in [1, n]$) captures a satisfying solution of Constraint (2).

The procedure *ExcludeCurTest* ensures that the explored solutions in test suite $\mathcal{T}$ are unique (*cf.* line 26). In particular, once $\mathcal{O}_{\mathbb{S}}$ is constructed from the execution trace $\mathbb{S}$, it modifies $\mathcal{O}_{\mathbb{S}}$ to exclude the previous solutions. For instance, if $\mathcal{T}$ includes solutions of exhibiting 1000 and 2000 cache misses, *objtrace* is modified to $\mathcal{O}_{\mathbb{S}} \wedge \sum_{i=1}^{n} miss_i \neq 1000 \wedge \sum_{i=1}^{n} miss_i \neq 2000$. Subsequently, this modified $\mathcal{O}_{\mathbb{S}}$ is leveraged to explore different solutions of the predicate Ω (*cf.* lines 31-34).

When $\Gamma(\Psi_{path}) \wedge \mathcal{O}_{\mathbb{S}} \wedge \Psi_{path}$ becomes unsatisfiable, *UnchkdPCs* keeps track of all unexplored partial path conditions (*cf.* lines 39-42) to manifest the remaining cache behaviour. In particular, our test generation satisfies the following crucial property.

Theorem 1. *`CATAPULT` guarantees to discover all possible cache behaviour upon termination. Besides, each input generated by `CATAPULT` witnesses a unique cache behaviour.*

4 Generating $\Gamma(\Psi_{path})$

Given a path condition Ψ_{path} and the execution trace $\mathbb{S}$ (*cf.* Equation (1)), this section describes the formulation of $\Gamma(\Psi_{path})$ – the set of all cache behaviour for inputs x

satisfying Ψ_{path} (*cf.* line 21 in Algorithm 1). In order to explain the formulation of $\Gamma(\Psi_{path})$, we assume the following notations throughout the paper:

- $2^{\mathcal{S}}$: The number of cache sets in the cache.
- $2^{\mathcal{B}}$: The size of a cache line (in bytes).
- $\mathcal{A}$: Associativity of cache.
- $set(r_i)$: Cache set accessed by memory-related instruction r_i.
- $tag(r_i)$: The tag stored in the cache for accessing address σ_i (*cf.* Equation (1)).
- ζ_i : The cache state before r_i and after r_{i-1}.

The formulation of $\Gamma(\Psi_{path})$ revolves around the concept of *cache conflict*. Formally, we define cache conflict as follows:

Definition 1 ***(Cache Conflict):*** *r_j generates a cache conflict to r_i only if r_j accesses a different memory block than r_i and executing r_j can influence the relative position of memory block $\lfloor \frac{\sigma_i}{2^{\mathcal{B}}} \rfloor$ within the cache state ζ_i.*

Clearly, r_j generates cache conflict to r_i only if $j < i$. In the next sections, we shall elaborate other crucial conditions required for the generation of cache conflicts. Subsequently, we build upon such conditions to formulate the number of cache misses.

4.1 Modeling Symbolic Cache Access

Recall from Equation (1) that we record the address σ_i (σ_i can be symbolic or concrete) for each memory-related instruction r_i during the execution. From σ_i, we formulate the accessed cache set $set(r_i)$ and the respective cache tag $tag(r_i)$ as follows:

$$set(r_i) = (\sigma_i \gg \mathcal{B}) \ \& \ \left(2^{\mathcal{S}} - 1\right); \qquad tag(r_i) = (\sigma_i \gg (\mathcal{B} + \mathcal{S})) \tag{3}$$

In Equation (3), "&" captures a bitwise-and operation and "$\gg$" captures a right-shift operation. Since σ_i can be symbolic, both $set(r_i)$ and $tag(r_i)$, as captured via Equation (3), can be symbolic expressions.

4.2 Modeling Symbolic Cache Constraints

In this section, we formulate constraints for the following two types of cache misses:

- *cold miss*: Cold miss occurs if a memory block is accessed for the *first time*.
- *eviction miss:* Any cache miss other than cold misses.

Conditions for cold misses If r_i accesses a memory block for the first time, the following condition must hold:

$$\Theta_i^{cold} \equiv \bigwedge_{1 \leq k < i} ((tag(r_k) \neq tag(r_i)) \vee (set(r_k) \neq set(r_i))) \tag{4}$$

Informally, Constraint (4) states that every memory access $r \in \{r_1, r_2, \ldots, r_{i-1}\}$ is either mapped to a different cache set than $set(r_i)$ or has a different tag compared to $tag(r_i)$. This leads to a cold cache miss at r_i.

In Constraint (4), for the sake of simplicity in the formulation, we assumed that initially, the cache is not loaded with any memory block used by the system under test. However, this condition can easily be relaxed via additional constraints that check the (un)availability of memory block $\lfloor \frac{\sigma_i}{2^{\mathcal{B}}} \rfloor$ in an arbitrary initial cache state.

Necessary conditions for cache conflict The basic design principle of cache dictates that every cache set is independent. Therefore, a necessary condition for cache conflict is that the accessed memory blocks are mapped to the same cache set. In particular, the following two conditions *must be satisfied* for a possible cache conflict from r_j to r_i:

1. $\psi_{cnf}(j,i)$: r_i and r_j access the same cache set. Therefore, we get the following:
$$\psi_{cnf}\ (j,i) \equiv (set(r_j) = set(r_i)) \tag{5}$$
2. $\psi_{dif}(j,i)$: r_i and r_j access different memory-block tags. Therefore, we have,
$$\psi_{dif}\ (j,i) \equiv (tag(r_j) \neq tag(r_i)) \tag{6}$$

The satisfiability of $\psi_{cnf}(j,i)$ and $\psi_{dif}(j,i)$ is necessary irrespective of the underlying cache replacement policy. However, these two constraints are not sufficient to guarantee that r_j can affect the cache state ζ_i $(i > j)$. We need additional constraints that depend on the specific replacement policy. In the subsequent sections, we formulate these constraints for two widely used replacement policies – LRU and FIFO.

Constraints for LRU caches In this section, we formulate a set of constraints that precisely capture the cache conflict scenarios in LRU replacement policy.

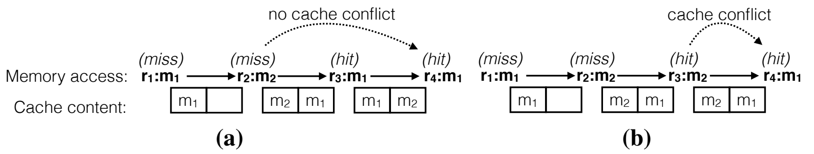


Fig. 4. Cache conflict scenarios for caches with LRU policy. r_i:m_j captures memory-related instruction r_i accessing memory block m_j. The rightmost position in the cache denotes the memory block *accessed* in the cache the *earliest*. (a) r_2 does not generate any cache conflict to r_4, as $m1$ is reloaded between r_2 and r_4, (b) in order to count unique cache conflicts to r_4, we only record the cache conflict from r_3 and not from r_2, as both r_2 and r_3 access m_2.

Conditions for eviction misses Let us check the conditions where instruction r_i will suffer a cache miss due to eviction. This might happen only due to instructions appearing before (in the program order) r_i. Consider one such instruction r_j, for $j \in [1, i)$. Informally, r_j generates a cache conflict to r_i, *only if* the following conditions hold:

1. $\psi_{eqv}^{lru}(j,i)$: There does not exist any instruction r_k where $k \in [j+1, i)$, such that r_k accesses the same memory block as r_i (*i.e.* $\lfloor \frac{\sigma_i}{2^{\mathcal{B}}} \rfloor$). It is worthwhile to note that the execution of r_k will make the memory block $\lfloor \frac{\sigma_i}{2^{\mathcal{B}}} \rfloor$ to be most recently used. For instance, in Figure 4(a), r_3 accesses memory block m_1 and therefore, r_2 cannot generate cache conflict to r_4. We capture $\psi_{eqv}(j,i)$ via the following constraints:
$$\psi_{eqv}^{lru}\ (j,i) \equiv \bigwedge_{k:\ j<k<i} ((tag(r_k) \neq tag(r_i)) \vee (set(r_k) \neq set(r_i))) \tag{7}$$

2. $\psi_{unq}^{lru}(j,i)$: Secondly, we need to count cache conflicts from unique memory blocks. As an example, consider the example shown in Figure 4(b). r_4 will still be a cache hit. This is because both r_2 and r_3 access the memory block m_2. In order to account unique cache conflicts, we only record the cache conflict from the *closest* access to different memory blocks. For instance, in Figure 4(b), we only record cache conflict from r_3 to r_4. We use the constraint $\psi_{unq}(j,i)$ for such purpose. $\psi_{unq}(j,i)$ is satisfiable if and only if there does not exist any memory-related instruction between r_j (where $j \in [1,i)$) and r_i that accesses the same memory block as r_j. Therefore, $\psi_{unq}(j,i)$ is captured as follows:

$$\psi_{unq}^{lru}(j,i) \equiv \bigwedge_{k:\ j<k<i} \left((tag(r_j) \neq tag(r_k)) \vee (set(r_j) \neq set(r_k))\right) \tag{8}$$

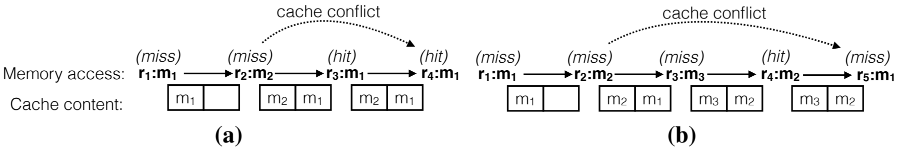
Constraints to Formulate Cache Conflict Constraints (5)-(8) accurately capture scenarios where r_j ($j \in [1,i)$) will create a unique cache conflict to r_i. Let us assume $\Psi_{i,j}^{evt}$ captures whether r_j creates a unique cache conflict to r_i. Using the intuition described in the preceding paragraph, we can now formulate the following constraints to set the value of $\Psi_{i,j}^{evt}$.

$$\Theta_{j,i}^{em,lru} \equiv \left(\psi_{cnf}(j,i) \wedge \psi_{dif}(j,i) \wedge \psi_{eqv}^{lru}(j,i) \wedge \psi_{unq}^{lru}(j,i)\right) \Rightarrow \left(\Psi_{j,i}^{evt} = 1\right) \tag{9}$$

If any of the conditions in Constraints (5)-(8) is not satisfied between r_j and r_i, then r_j cannot influence the cache state immediately before r_i and therefore, r_j cannot create cache conflict to r_i, as captured by the following constraints:

$$\Theta_{j,i}^{eh,lru} \equiv \left(\neg\psi_{cnf}(j,i) \vee \neg\psi_{dif}(j,i) \vee \neg\psi_{eqv}^{lru}(j,i) \vee \neg\, \psi_{unq}^{lru}(j,i)\right) \Rightarrow \left(\Psi_{j,i}^{evt} = 0\right) \tag{10}$$

Constraints for FIFO Caches Unlike LRU replacement policy, for FIFO replacement policy, the cache state does not change for a cache hit. Therefore, r_j can generate a cache conflict to r_i (where $i > j$) only if r_j is a cache miss.



(a) **(b)**

Fig. 5. Cache conflict scenarios in FIFO policy. r_i:m_j captures memory-related instruction r_i accessing memory block m_j. The rightmost position in the cache denotes the memory block *inserted* in the cache the *earliest*. (a) r_2 generates cache conflict to r_4 even though m_1 is accessed at r_3. This is because r_3 is a cache hit. (b) We record cache conflict from r_2 to r_5 even though r_4 is closer to r_5 and r_5 accesses the same memory block as r_2. This is because r_4 is a cache hit.

Conditions for eviction misses In order to incorporate the changes in the formulation of $\Gamma(\Psi_{path})$, we need to modify Constraints (7)-(10) for FIFO replacement policy. In particular, instruction r_j can create a unique cache conflict to instruction r_i ($i > j$) only if r_j is a cache miss and the following conditions hold with $\psi_{cnf}(j,i)$ and $\psi_{dif}(j,i)$:

1. $\psi_{eqv}^{fifo}(j,i)$: There does not exist any instruction r_k, where $k \in [j+1, i)$, such that r_k is a cache miss and it accesses the same memory block as r_i. For instance, in Figure 5(a), r_2 generates cache conflict to r_4 because r_3 was a cache hit. We capture $\psi_{eqv}^{fifo}(j,i)$ as follows:

$$\psi_{eqv}^{fifo}(j,i) \equiv \bigwedge_{k:\ j<k<i} ((tag(r_k) \neq tag(r_i)) \vee (set(r_k) \neq set(r_i)) \vee (miss_k = 0)) \tag{11}$$

2. $\psi_{unq}^{fifo}(j,i)$: This constraint ensures that we only count unique cache conflicts. For LRU policy, we checked whether r_j was the closest instruction to r_i accessing memory block $\lfloor \frac{\sigma_j}{2^{\mathcal{B}}} \rfloor$. For FIFO policy, we have a slightly different situation, as demonstrated in Figure 5(b). Even though r_4 is the closest instruction to r_5 accessing m_2, r_4 cannot generate cache conflict to r_5. This is because r_4 is a cache hit. As a result, we record cache conflict from r_2 to r_5. It is worthwhile to mention that in LRU policy, we will discard the cache conflict from r_2 to r_5 due to the presence of r_4. Formally, we ensure there does not exist any instruction r_k, where $k \in [j+1, i)$, such that r_k is a cache miss and it accesses the same memory block as r_j. Therefore, $\psi_{unq}^{fifo}(j,i)$ can be formalized as follows:

$$\psi_{unq}^{fifo}(j,i) \equiv \bigwedge_{k:\ j<k<i} ((tag(r_j) \neq tag(r_k)) \vee (set(r_j) \neq set(r_k)) \vee (miss_k = 0)) \tag{12}$$

Constraints to formulate cache conflict Let us assume $\Psi_{j,i}^{evt}$ captures whether r_j creates a cache conflict to r_i. For FIFO replacement policy, this is possible only if r_j is a cache miss (*i.e.* $miss_j = 1$). Using the intuition described in the preceding paragraphs, we can bound the value of $\Psi_{j,i}^{evt}$ as follows:

$$\Theta_{j,i}^{em,fifo} \equiv \left(\psi_{cnf}(j,i) \wedge \psi_{dif}(j,i) \wedge \psi_{eqv}^{fifo}(j,i) \wedge \psi_{unq}^{fifo}(j,i) \wedge (miss_j = 1)\right) \Rightarrow \left(\Psi_{j,i}^{evt} = 1\right) \tag{13}$$

$$\Theta_{j,i}^{eh,fifo} \equiv \left(\neg\psi_{cnf}(j,i) \vee \neg\psi_{dif}(j,i) \vee \neg\psi_{eqv}^{fifo}(j,i) \vee \neg\psi_{unq}^{fifo}(j,i) \vee (miss_j = 0)\right) \Rightarrow \left(\Psi_{j,i}^{evt} = 0\right) \tag{14}$$

Constraints to formulate cache misses Let us assume that $miss_i$ captures the cache behaviour of instruction r_i. Therefore, $miss_i$ is set to 1 if r_i is a cache miss, and is set to 0 otherwise. We can formulate the value of $miss_i$ using the following constraints:

$$\Theta_i^{mp} \equiv \left(\sum_{j \in [1,i)} \Psi_{j,i}^{evt} \geq \mathcal{A}\right) \vee \Theta_i^{cold} \tag{15}$$

$$\Theta_i^{m} \equiv \Theta_i^{mp} \Rightarrow (miss_i = 1)\,; \;\; \Theta_i^{h} \equiv \neg\Theta_i^{mp} \Rightarrow (miss_i = 0) \tag{16}$$

where $\mathcal{A}$ captures the associativity of the cache. Once a memory block is loaded into the cache, it requires at least $\mathcal{A}$ unique cache conflicts to evict the block. If $\Psi_{i,j}^{evt} \geq \mathcal{A}$, r_i has suffered at least $\mathcal{A}$ unique cache conflicts since the last access of the memory block referenced by r_i – resulting r_i to be a cache miss. If r_i is not a cold miss (*i.e.* $\neg\Theta_i^{cold}$ holds) and $\Psi_{i,j}^{evt} \geq \mathcal{A}$ does not hold, r_i will be a cache hit, as captured by Constraint (16).

Putting it all together To derive the symbolic cache behavior $\Gamma(\Psi_{path})$, we gather all constraints over $\{r_1, \ldots, r_n\}$ as follows:

$$\boxed{\Gamma(\Psi_{path}) \equiv \bigwedge_{i \in [1,n]} \left(\Theta_i^m \wedge \Theta_i^h \wedge \bigwedge_{j \in [1,i)} \Theta_{j,i}^{em,repl} \wedge \bigwedge_{j \in [1,i)} \Theta_{j,i}^{eh,repl} \right)} \tag{17}$$

where $repl \in \{lru, fifo\}$ capturing the underlying replacement policy. Θ_i^m and Θ_i^h together bound the value of $miss_i$, which, in turn captures whether r_i is a cache miss. However, Θ_i^m and Θ_i^h are dependent on symbolic variables $\Psi_{j,i}^{evt}$ where $j \in [1, i)$. The bound on symbolic variables $\Psi_{j,i}^{evt}$ is captured via $\Theta_{j,i}^{em,repl}$ and $\Theta_{j,i}^{eh,repl}$ (Constraints (9)-(10) and Constraints (13)-(14)). Hence, the formulation of $\Gamma(\Psi_{path})$ includes both $\Theta_{j,i}^{em,repl}$ and $\Theta_{j,i}^{eh,repl}$ for $j \in [1, i)$.

Complexity of constraints The size of our constraint system is $O(n^3)$, where n is the number of memory accesses. The dominating factor in our constraint system is the set of constraints generated from Constraints (9)-(10) for LRU policy and from Constraints (13)-(14) for FIFO policy. In general, we generate constraints for each pair of memory accesses that may potentially conflict in the cache, leading to $O(n^2)$ pairs in total. For each such pair, the constraint may have a size $O(n)$ — making the size of overall constraint system to be $O(n^3)$. However, our evaluation reveals that such a bound is pessimistic and the constraint system can be solved efficiently for real-life programs.

5 Application

In this section, we instantiate Algorithm 1 to formulate the objective $\mathcal{O}_{\mathbb{S}}$ from the execution trace $\mathbb{S}$ and the refinement of $\mathcal{O}_{\mathbb{S}}$ (*cf.* line 23 and lines 31-34 in Algorithm 1).

Testing timing-related properties Embedded and real-time systems are often constrained via several timing-related properties. Given a timing deadline $\mathcal{D}$, Algorithm 1 can find a witness where such timing deadline is violated for program $\mathcal{P}$ or prove that no such witness exists.

In this paper, we assume that the timing of a given instruction may vary only due to the same incurring a cache hit or a cache miss. However, such a timing model can always be extended leveraging on the rich body of work in timing analysis [21].

Given the execution trace $\mathbb{S}$ (*cf.* Equation (1)), we use the variable $miss_i$ to capture whether a memory-related instruction r_i suffered a cache miss. Let us assume $\mathbb{C}$ is the time taken to execute all instructions not accessing the memory subsystems. Given the preceding descriptions, we formulate the objective $\mathcal{O}_{\mathbb{S}}$ from $\mathbb{S}$ as follows:

$$\boxed{\mathcal{O}_{\mathbb{S}} \equiv \left(\sum_{i=1}^{n} miss_i \right) * \mathcal{L} + \mathbb{C} > \mathcal{D}} \tag{18}$$

where $\mathcal{L}$ is the latency incurred for a cache miss and n is the total number of memory-related instructions. If a solution is found for $\Gamma(\Psi_{path}) \wedge \mathcal{O}_{\mathbb{S}} \wedge \Psi_{path}$ using $\mathcal{O}_{\mathbb{S}}$ in Equation (18), then we found witness of a violation of timing deadline $\mathcal{D}$. Such a witness can be used for further investigation and improve the timing behaviour of the system.

In our evaluation, we refine $\mathcal{O}_{\mathbb{S}}$ to find unique violations, meaning each test input capture a unique value of $\sum_{i=1}^{n} miss_i * \mathcal{L} + \mathbb{C}$. Therefore, if $\sum_{i=1}^{n} miss_i = N$ is true for a satisfying solution of $\Gamma\left(\Psi_{path}\right) \wedge \mathcal{O}_{\mathbb{S}} \wedge \Psi_{path}$, $\mathcal{O}_{\mathbb{S}}$ is refined as $\mathcal{O}_{\mathbb{S}} \wedge \sum_{i=1}^{n} miss_i \neq N$.

Testing cache side-channel vulnerabilities The performance gap between cache and main memory (DRAM) can be exploited by an attacker to discover classified information (*e.g.* a secret key). Such attacks are often non-invasive and they can even be mounted over the network [8]. In this paper, we choose timing-related attacks, where the observer monitors the overall cache misses to discover secret information [8].

Let us assume the cache side channel to be a function $C : \mathbb{I} \rightarrow \mathbb{O}$, mapping a finite set of secret inputs to a finite set of observations. Since the attacker monitors the number of cache misses, in this scenario, an observation simply captures the number of cache misses in an execution. If we model the choice of a secret input via a random variable X and the respective observation by a random variable Y, the leakage through channel C is the reduction in uncertainty about X when Y is observed. In particular, the following result holds for any distribution of X [17].

$$ML(C) \leq \log_2 |C(\mathbb{I})| \tag{19}$$

where $ML(C)$ captures the maximal leakage of channel C. The equality holds in Equation (19) when X is uniformly distributed.

CATAPULT can be tuned to compute each unique element in the set $C(\mathbb{I})$ and thereby, to derive an upper bound (exact bound when X is uniformly distributed) on the maximal leakage $ML(C)$. We accomplish this by setting and refining $\mathcal{O}_{\mathbb{S}}$ as follows:

$$\boxed{\mathcal{O}_{\mathbb{S}} \equiv \left(\sum_{i=1}^{n} miss_i \geq 0\right)} \tag{20}$$

If $miss_i^{(c)}$ captures a satisfying solution of $miss_i$ (for $i \in [1, n]$) in $\Gamma\left(\Psi_{path}\right) \wedge \mathcal{O}_{\mathbb{S}} \wedge \Psi_{path}$, then we refine $\mathcal{O}_{\mathbb{S}}$ as follows: $\mathcal{O}_{\mathbb{S}} \wedge \left(\sum_{i=1}^{n} miss_i \neq \sum_{i=1}^{n} miss_i^{(c)}\right)$.

It is worthwhile to mention that the number of tests computed is directly correlated with the maximal leakage through the cache side channel (*cf.* Equation (19)). As a result, our test generation method can be used as a metric to measure the information leak through cache side channel. Besides, since we also generate an witness for each possible observation (*i.e.* the number of cache misses), these witnesses can further be used for analyzing, quantifying and controlling the information leaked at runtime.

Due to the lack of space, we only show the instantiation for one type of attacker. However, our framework can model a variety of different attacking scenarios, as long as the observation by an attacker can be modeled via symbolic constraints over the set of variables $\{miss_1, miss_2, \ldots, miss_n\}$.

6 Evaluation

Experimental setup We build CATAPULT on top of KLEE symbolic execution engine [2]. We first decompile PISA [5] compliant binaries (a MIPS like architecture) into

LLVM bitcode. It is worthwhile to note that compiling *source code* to LLVM bitcode will inaccurately capture the cache performance. This is because of the target-dependent compiler optimizations that take place while generating binary code. The decompiled LLVM bitcode is identical with the PISA binary in terms of functionality, memory placement and the number of memory-related instructions. This ensures that the translated LLVM code has exactly the same cache performance as the binary code. To use CATAPULT for a different architecture (*e.g.* ARM), we only need the translator that converts the binary code for the respective architecture to the LLVM bitcode. The rest of our test generation framework remains completely unchanged. The translated LLVM code is provided as an input to CATAPULT. All our experiments have been performed on an Intel I7 machine with 8GB of RAM and running Debian operating system.

To evaluate CATAPULT, we choose cryptographic routines from OpenSSL and other libraries [3,1] and user-interface routines from Linux GDK library (*cf.* Table 1). Our choice is motivated by the importance of validating security and performance related properties in these programs. Moreover, these programs are memory intensive and in particular, the cryptographic routines exhibit complex memory access patterns. As a result, such programs are also appropriate for stress testing our framework.

Program name	Input size	Lines of C code	Lines of LLVM code	Max. no. of memory accesses
AES [1]	16 bytes	800	4950	2134
AES [3]	16 bytes	1428	1800	420
DES [3]	8 bytes	552	3990	334
RC4 [3]	10 bytes	160	668	1538
RC5 [3]	16 bytes	256	1820	410
gdk_keyval_to_unicode	4 bytes	1300	268	114
gdk_keyval_name	4 bytes	1350	1408	12

Table 1. Evaluated subject programs (input sizes are unchanged from the original programs)

Program	replacement policy	#tests	Time (in cycles) [min,max]	maximum no. of constraints	Testing time
AES [1]	LRU	35	[3719,7619]	2397228	260 min
	FIFO	1	[5149,5149]	11578752	15 sec
AES [3]	LRU	37	[1796,4996]	26528	127 min
	FIFO	1	[1896,1896]	1205860	3 min
DES [3]	LRU	21	[3971,6071]	1501080	10 min
	FIFO	1	[7971,7971]	1947656	2 sec
RC4 [3]	LRU	1	[5553,5553]	337588	15 min
	FIFO	1	[3153,3153]	764208	15 sec
RC5 [3]	LRU	1	[6167,6167]	0	10 sec
	FIFO	1	[6367,6367]	0	10 sec
gdk_keyval_to_unicode	LRU	19	[652,2652]	10	13 sec
	FIFO	28	[652,4852]	10	12 sec
gdk_keyval_name	LRU	11	[126,1126]	11	18 sec
	FIFO	11	[126,1126]	11	18 sec

Table 2. "#test" captures the total number of tests generated, where each test exhibits a unique cache performance (*cf.* section 5). Testing time includes the total time to run Algorithm 1.

Basic result Table 2 captures the key result obtained from CATAPULT. For all experiments in Table 2, we used a two-way and 8 KB cache, with 32 bytes cache-line and a cache-miss latency of 10 cycles. We make the following crucial observations from Table 2. We observe that the number of tests generated for FIFO policy is significantly smaller than the number of tests obtained for LRU policy. Since each test is attached to a unique cache performance (*i.e.* the number of cache miss), the LRU policy suffers from significantly higher cache side-channel leakage (*cf.* Equation (19)), as compared to FIFO policy. This happens due to the reason that cache states change more frequently in LRU policy as compared to FIFO policy (*e.g.* for every access in LRU policy and for every cache miss in FIFO policy). This potentially leads to more variation in cache performance across different inputs in LRU policy, resulting in more tests and higher cache side-channel leakage. This result indicates important trade-offs in system design, as LRU policy is, on average, superior compared to FIFO policy in terms of overall performance. For experiments in Table 2, we can also validate that the maximal leakage of a basic AES implementation [3] is comparable with the AES implementation in the OpenSSL library. The implementation of `RC5` does not exhibit any input-dependent memory access or branch. Hence, the size of our constraint system is 0 and there is exactly one test generated for both replacement policies. As observed from Table 2, `CATAPULT` terminates within reasonable time for all the experiments. Therefore, our results guarantee both the exact number of test cases and the best/worst-case timing obtained over all inputs. Finally, it is worthwhile to note that an exhaustive enumeration (2^{128} possibilities for AES) of all test inputs is infeasible to provide such guarantees.

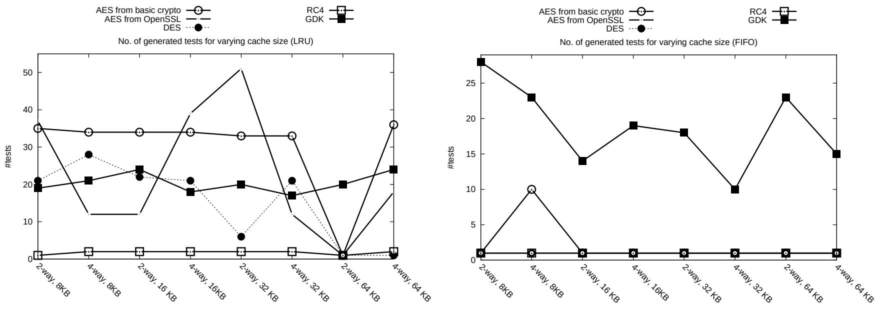


Fig. 6. Number of tests w.r.t. different cache configurations

Sensitivity result Figure 6 shows the sensitivity of `CATAPULT` with respect to cache configurations. Although increasing cache size usually improves performance, this may not be true for security. As an example, a smaller cache may result in cache misses for all possible inputs (*i.e.* one test), whereas a bigger cache may result in cache misses for a subset of inputs (*i.e.* more than one test). As a result, increasing the cache size may lead to increased number of tests and hence, increased likelihood of cache side-channel leakage (*cf.* Equation (19)). For a huge cache, however, the dependency between inputs and the cache behaviour may disappear, resulting in reduced cache side-channel leakage. In Figure 6, we observe both the increase and the decrease in the number of tests (and hence, the maximal leakage) with increased cache size. We also observe that FIFO policy on average outperforms LRU policy, in terms of side-channel resistant.

Summary In summary, `CATAPULT` reveals useful information regarding the memory performance of programs. This includes the cache side-channel vulnerability of these programs as well as their worst-case memory performance. Concretely, we can show, for most of the chosen subjects, FIFO replacement policy is significantly more resistant to cache side channel as compared to LRU policy. We also show that increasing cache size may not necessarily lead to a more secure implementation (*cf.* Figure 6).

7 Related Work

Works on worst-case execution time (WCET) analysis [20,21] compute an upper bound on the execution time of program. In addition, approaches based on program synthesis [9] aim to generate optimal software by construction. In contrast to these works, our approach has a significant flavor of testing and CATAPULT is used to generate witnesses violating certain non-functional properties. Our work is orthogonal to the efforts in statically analyzing cache side channels [17,14,7]. Specifically, CATAPULT generates test inputs that violate timing-related constraints and to quantify cache side-channel leakage. Our framework does not generate false positives, however, it is not aimed to verify the absence of cache side-channel leaks and timing-related violations.

Recent works on performance testing [16,18,19] consider performance purely at code level and ignore any influence of execution platform on performance. Our previous works had targeted specific performance bugs (*e.g.* cache thrashing [6]) and they do not consider the variation of memory performance within a program path [6,13,11].

In summary, a majority of works in software testing have either focused on functionality bugs or ignore the influence of execution platforms on non-functional properties. In this paper, we propose to systematically explore the behaviour of execution platform via dynamic symbolic execution, with a specific focus on memory performance.

8 Discussion

Extensions and limitations CATAPULT generates witnesses to show the evidence of side-channel leakage or timing-related faults. However, it does not debug these faults. Debugging faults related to software non-functional properties (*e.g.* timing and security) is a challenging research problem in its own right and CATAPULT provides the necessary foundation for debugging research in the context of timing or security-related faults. A limitation of our approach is the requirement of the knowledge of cache architecture (*e.g.* cache replacement policy). In the future, this limitation can be lifted via using some machine learning approach to approximately capture the cache replacement policy [4]. Subsequently, we can formulate the test generation problem via symbolic constraints in a similar fashion as in CATAPULT. The scalability of CATAPULT is primarily limited by the number of memory accesses in an execution. Since our symbolic constraints encode the cache conflict (and not the actual cache states), the scalability of CATAPULT is not affected by increased cache sizes and associativity [10].

Perspective We have presented CATAPULT where the key insight is to express the platform-dependent software properties (*e.g.* performance and security) via logical constraints. Hence, our approach can be adapted easily within existing software testing methodologies based on symbolic execution. To show the generality of our approach, we have instantiated our framework for a variety of cache designs and shown its application in both performance and security testing via real-life case studies (*e.g.* including OpenSSL and Linux GDK applications). This makes the idea of CATAPULT quite appealing for further exploration in the future. Among others, techniques to improve the testing time and extension of CATAPULT for regression testing are worth exploring in the area of testing non-functional software properties.

Acknowledgement We thank Ezekiel Olamide Soremekun and the anonymous reviewers for insightful feedback on an earlier version of this paper.

References

1. Advanced Encryption Standard Implementation. `https://github.com/B-Con/crypto-algorithms`.
2. KLEE LLVM execution engine. `https://klee.github.io/`.
3. OpenSSL Library. `https://github.com/openssl/openssl/tree/master/crypto`.
4. Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *RTAS*, pages 65–74, 2013.
5. Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2), 2002.
6. Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis driven cache performance testing. In *RTSS*, pages 319–329, 2013.
7. Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014.
8. Daniel J Bernstein. Cache-timing attacks on AES, 2005.
9. Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *CAV*, pages 243–259, 2011.
10. Sudipta Chattopadhyay. Directed Automated Memory Performance Testing. `http://sudiptac.bitbucket.org/papers/catapult-TR.pdf`.
11. Sudipta Chattopadhyay. MESS: memory performance debugging on embedded multi-core systems. In *SPIN*, pages 105–125, 2015.
12. Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leak in cache attacks through symbolic execution. *CoRR*, abs/1611.04426, 2016.
13. Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. Automated software testing of memory performance in embedded gpus. In *EMSOFT*, pages 17:1–17:10, 2014.
14. Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: a tool for the static analysis of cache side channels. *TISSEC*, 18(1):4, 2015.
15. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, 2005.
16. Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, 2012.
17. Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *CAV*. Springer, 2012.
18. Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.
19. Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, pages 369–378, 2015.
20. Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3), 2000.
21. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

Appendix

In this section, we illustrate the construction of our symbolic cache model for a **two-way set-associative cache**. Let us assume that we execute the code in Figure 2(a) for input $x = 0$. This results in the execution of the *if* branch. After the execution, we record the path condition and the execution trace as follows:

$$\Psi_{path} \equiv (x \geq 0 \wedge x \leq 127) \tag{22}$$

$$\mathbb{S} \equiv \langle (r_1, \&p + 127 - x), (r_2, \&q + x), (r_3, \&p + 127 - x) \rangle \tag{23}$$

In an abuse of notation, we use $@R$ to capture the memory block containing address $\&R$. For instance, $@p[127 - x]$ captures the memory block containing address $\&p + 127 - x$. Recall that we use $miss_i$ to capture whether the instruction r_i suffers a cache miss. Starting with an empty cache, r_1 is a cold miss and therefore, $miss_1 = 1$. In the following, we formulate the constraints to bound $miss_2$ and $miss_3$.

Constraints to bound $\boldsymbol{miss_2}$

In order to check whether r_2 is a cold miss, we need to check that r_2 accesses the memory block $@q[x]$ for the first time. This can be accomplished by examining the satisfiability of the following constraint (*cf.* Constraint (5)):

$$\Theta_2^{cold} \equiv (tag(r_1) \neq tag(r_2)) \vee (set(r_1) \neq set(r_2)) \tag{24}$$

Since only r_1 is accessed before r_2, only r_1 can potentially generate cache conflict to r_2. The following constraints must be satisfied if r_1 generates cache conflict to r_2 (*cf.* Constraints (6)-(7)).

$$\psi_{cnf}(1, 2) \equiv set(r_1) = set(r_2) \tag{25}$$

$$\psi_{dif}(1, 2) \equiv tag(r_1) \neq tag(r_2) \tag{26}$$

Constraints for LRU caches Since there does not exist any memory access between r_1 and r_2, memory blocks $@q[x]$ and $@p[127 - x]$ cannot be accessed between r_1 and r_2. Therefore, both $\psi_{eqv}^{lru}(1, 2)$ and $\psi_{unq}^{lru}(1, 2)$ are trivially evaluated to *true* (*cf.* Constraints (8)-(9)). As a result, we bound the value of $miss_2$ by combining constraints (24)-(26) as follows (*cf.* Constraints (10)-(11) and Constraints (16)-(17)):

$$\psi_{cnf}(1, 2) \wedge \psi_{dif}(1, 2) \Rightarrow \left(\Psi_{1,2}^{evt} = 1\right) \tag{27}$$

$$\neg\psi_{cnf}(1, 2) \vee \neg\psi_{dif}(1, 2) \Rightarrow \left(\Psi_{1,2}^{evt} = 0\right) \tag{28}$$

$$\left(\Psi_{1,2}^{evt} \geq 2\right) \vee \Theta_2^{cold} \Rightarrow (miss_2 = 1) \tag{29}$$

$$\left(\Psi_{1,2}^{evt} < 2\right) \wedge \neg\Theta_2^{cold} \Rightarrow (miss_2 = 0) \tag{30}$$

Constraints for FIFO caches For caches with FIFO replacement policy, $\psi_{eqv}^{fifo}(1,2)$ and $\psi_{unq}^{fifo}(1,2)$ are also evaluated to *true* (*cf.* Constraints (12)-(13)), as there does not exist any memory access between r_1 and r_2. However, since FIFO cache states are only changed during a cache miss, we need to modify constraints (27)-(28) as follows (*cf.* Constraints (14)-(15):

$$\psi_{cnf}(1,2) \wedge \psi_{dif}(1,2) \wedge (miss_1 = 1) \Rightarrow \left(\Psi_{1,2}^{evt} = 1\right) \tag{31}$$

$$\neg\psi_{cnf}(1,2) \vee \neg\psi_{dif}(1,2) \vee (miss_1 = 0) \Rightarrow \left(\Psi_{1,2}^{evt} = 0\right) \tag{32}$$

$$\left(\Psi_{1,2}^{evt} \geq 2\right) \vee \Theta_2^{cold} \Rightarrow (miss_2 = 1) \tag{33}$$

$$\left(\Psi_{1,2}^{evt} < 2\right) \wedge \neg\Theta_2^{cold} \Rightarrow (miss_2 = 0) \tag{34}$$

Constraints to bound $miss_3$

In order to check whether r_3 is a cold miss, we need to ensure neither r_1 nor r_2 loads the memory block $@p[127-x]$. This can be done by checking the satisfiability of the following constraint:

$$\begin{aligned}\Theta_3^{cold} \equiv\ & (tag(r_1) \neq tag(r_3) \vee set(r_1) \neq set(r_3)) \wedge \\ & (tag(r_2) \neq tag(r_3) \vee set(r_2) \neq set(r_3))\end{aligned} \tag{35}$$

It is worthwhile to note that Θ_3^{cold} is *unsatisfiable*. This is because r_1 and r_3 both accesses the same memory block $@p[127-x]$ and therefore, both $tag(r_1) \neq tag(r_3)$ and $set(r_1) \neq set(r_3)$ are evaluated to *false*.

Necessary conditions for cache conflict Both r_1 and r_2 may generate cache conflict to r_3. The following necessary conditions must hold for r_1 and r_2 to generate cache conflict to r_3:

$$\psi_{cnf}(1,3) \equiv set(r_1) = set(r_3) \tag{36}$$

$$\psi_{dif}(1,3) \equiv tag(r_1) \neq tag(r_3) \tag{37}$$

$$\psi_{cnf}(2,3) \equiv set(r_2) = set(r_3) \tag{38}$$

$$\psi_{dif}(2,3) \equiv tag(r_2) \neq tag(r_3) \tag{39}$$

Constraints for LRU caches Since r_2 is accessed between r_1 and r_3, we need to ensure that r_2 does not reload memory block accessed by r_3 (*i.e.* $@p[127-x]$). This is accomplished via the following constraint:

$$\psi_{eqv}^{lru}(1,3) \equiv (tag(r_2) \neq tag(r_3) \vee set(r_2) \neq set(r_3)) \tag{40}$$

Finally, we check that r_2 does not access the same memory block accessed by r_1 (*i.e.* $@p[127-x]$). This is to avoid counting duplicate cache conflicts from the same memory block. $\psi_{unq}^{lru}(1,3)$ ensures that r_1 does not access the same memory block accessed by r_2.

$$\psi_{unq}^{lru}(1,3) \equiv (tag(r_1) \neq tag(r_2) \vee set(r_1) \neq set(r_2)) \tag{41}$$

It is worthwhile to note that both $\psi_{unq}^{lru}(2,3)$ and $\psi_{eqv}^{lru}(2,3)$ are trivially *true*, as there does not exist any memory access between r_2 and r_3.

In order to bound the value of $miss_3$, we gather the aforementioned constraints in the following formulation:

$$\psi_{cnf}(1,3) \wedge \psi_{dif}(1,3) \wedge \psi^{lru}_{eqv}(1,3) \wedge \psi^{lru}_{unq}(1,3) \Rightarrow \left(\Psi^{evt}_{1,3} = 1\right) \tag{42}$$

$$\neg\psi_{cnf}(1,3) \vee \neg\psi_{dif}(1,3) \vee \neg\psi^{lru}_{eqv}(1,3) \vee \neg\psi^{lru}_{unq}(1,3) \Rightarrow \left(\Psi^{evt}_{1,3} = 0\right) \tag{43}$$

$$\psi_{cnf}(2,3) \wedge \psi_{dif}(2,3) \Rightarrow \left(\Psi^{evt}_{2,3} = 1\right) \tag{44}$$

$$\neg\psi_{cnf}(2,3) \vee \neg\psi_{dif}(2,3) \Rightarrow \left(\Psi^{evt}_{2,3} = 0\right) \tag{45}$$

$$\left(\Psi^{evt}_{1,3} + \Psi^{evt}_{2,3} \geq 2\right) \vee \Theta^{cold}_{3} \Rightarrow (miss_3 = 1) \tag{46}$$

$$\left(\Psi^{evt}_{1,3} + \Psi^{evt}_{2,3} < 2\right) \wedge \neg\Theta^{cold}_{3} \Rightarrow (miss_3 = 0) \tag{47}$$

Constraints for FIFO caches As explained in the construction of $\psi^{fifo}_{eqv}(i,j)$ and $\psi^{fifo}_{unq}(i,j)$ (*cf.* Constraints (12)-(13)), we need additional constraints to reflect that cache states are only changed for cache misses. In particular, $\psi^{fifo}_{eqv}(1,3)$ and $\psi^{fifo}_{eqv}(1,3)$ are formulated as follows:

$$\psi^{fifo}_{eqv}(1,3) \equiv (tag(r_2) \neq tag(r_3) \vee set(r_2) \neq set(r_3) \vee miss_2 = 0) \tag{48}$$

$$\psi^{fifo}_{unq}(1,3) \equiv (tag(r_1) \neq tag(r_2) \vee set(r_1) \neq set(r_2) \vee miss_2 = 0) \tag{49}$$

Like LRU policy, we note that both $\psi^{fifo}_{unq}(2,3)$ and $\psi^{fifo}_{eqv}(2,3)$ are trivially *true*, as there does not exist any memory access between r_2 and r_3.

Finally, we record cache conflict from r_1 (or r_2) to r_3 only if r_1 (or r_2) is a cache miss, meaning $miss_1 = 1$ (or $miss_2 = 1$). This is accomplished via the following constraints:

$$\psi_{cnf}(1,3) \wedge \psi_{dif}(1,3) \wedge \psi^{fifo}_{eqv}(1,3) \wedge \psi^{fifo}_{unq}(1,3) \wedge (miss_1 = 1) \Rightarrow \left(\Psi^{evt}_{1,3} = 1\right) \tag{50}$$

$$\neg\psi_{cnf}(1,3) \vee \neg\psi_{dif}(1,3) \vee \neg\psi^{fifo}_{eqv}(1,3) \vee \neg\psi^{fifo}_{unq}(1,3) \vee (miss_1 = 0) \Rightarrow \left(\Psi^{evt}_{1,3} = 0\right) \tag{51}$$

$$\psi_{cnf}(2,3) \wedge \psi_{dif}(2,3) \wedge (miss_2 = 1) \Rightarrow \left(\Psi^{evt}_{2,3} = 1\right) \tag{52}$$

$$\neg\psi_{cnf}(2,3) \vee \neg\psi_{dif}(2,3) \vee (miss_2 = 0) \Rightarrow \left(\Psi^{evt}_{2,3} = 0\right) \tag{53}$$

$$\left(\Psi^{evt}_{1,3} + \Psi^{evt}_{2,3} \geq 2\right) \vee \Theta^{cold}_{3} \Rightarrow (miss_3 = 1) \tag{54}$$

$$\left(\Psi^{evt}_{1,3} + \Psi^{evt}_{2,3} < 2\right) \wedge \neg\Theta^{cold}_{3} \Rightarrow (miss_3 = 0) \tag{55}$$

Summary Constraints to bound $miss_2$ and $miss_3$, together with the fact $miss_1 = 1$ capture our symbolic cache model $\Gamma(x \geq 0 \wedge x \leq 127)$. This model is explored during symbolic execution to discover all possible cache behaviour in a program.

Test Generation Time

Tables 3-4 report the time needed to generate all tests using `CATAPULT`.

Program name	Testing time (in seconds)						
	4-way, 8KB	2-way, 16KB	4-way, 16KB	2-way, 32KB	4-way, 32KB	2-way, 64KB	4-way, 64KB
AES [1]	22917	1720	16971	2006	2190	496	2581
AES [3]	442	484	4125	7409	472	98	626
DES [3]	11193	2657	762	138	376	134	130
RC4 [3]	1030	993	989	955	995	888	952
RC5 [3]	14	14	14	14	14	14	14
gdk_keyval_to_unicode	14	13	14	14	14	14	14
gdk_keyval_name	18	18	17	18	18	18	19

Table 3. Testing time for LRU replacement policy

Program name	Testing time (in seconds)						
	4-way, 8KB	2-way, 16KB	4-way, 16KB	2-way, 32KB	4-way, 32KB	2-way, 64KB	4-way, 64KB
AES [1]	15	15	68	15	14	14	14
AES [3]	750	14	209	2	14	2	2
DES [3]	56	2	2	2	2	2	889
RC4 [3]	16	16	68	18	15	15	18
RC5 [3]	14	15	15	15	15	15	15
gdk_keyval_to_unicode	13	14	13	14	14	14	14
gdk_keyval_name	18	18	18	19	18	18	18

Table 4. Testing time for FIFO replacement policy