APPENDIX

The appendix includes additional cache models (e.g. LRU and FIFO) incorporated within CACHEFIX, the theoretical guarantees and additional experimental results.

Theoretical Guarantees

In this section, we include the detailed proof of the properties satisfied by CACHEFIX.

PROPERTY 3. ***(Monotonicity)*** *Consider a victim program $\mathcal{P}$ with sensitive input $\mathcal{K}$. Given attack models $\mathcal{O}_{time}$ or $\mathcal{O}_{trace}$, assume that the channel capacity to quantify the uncertainty of guessing $\mathcal{K}$ is $\mathcal{G}^{\mathcal{P}}_{cap}$. Our checker guarantees that $\mathcal{G}^{\mathcal{P}}_{cap}$ monotonically decreases with each synthesized patch being applied at runtime.*

PROOF. Consider the generic attack model $\mathcal{O} : \{h, m\}^* \rightarrow \mathbb{X}$ that maps each trace to an element in the countable set $\mathbb{X}$. For a victim program $\mathcal{P}$, assume $TR \subseteq \{h, m\}^*$ is the set of all execution traces. After one round of patch synthesis, assume $TR' \subseteq \{h, m\}^*$ is the set of all execution traces in $\mathcal{P}$ when the synthesized patches are applied at runtime. By construction, each round of patch synthesis merges two equivalence classes of observations (*cf.* Algorithm 3), hence, making them indistinguishable by the attacker $\mathcal{O}$. As a result, the following relationship holds:

$$|\mathcal{O}(TR)| = \left|\mathcal{O}(TR')\right| + 1 \tag{29}$$

Channel capacity $\mathcal{G}^{\mathcal{P}}_{cap}$ equals to $\log |\mathcal{O}(TR)|$ for the original program $\mathcal{P}$, but it reduces to $\log |\mathcal{O}(TR')|$ when the synthesized patches are applied. We conclude the proof as the same argument holds for any round of patch synthesis. □

PROPERTY 4. ***(Convergence)*** *Let us assume a victim program $\mathcal{P}$ with sensitive input $\mathcal{K}$. In the absence of any attacker, assume that the uncertainty to guess $\mathcal{K}$ is $\mathcal{G}^{init}_{cap}$, $\mathcal{G}^{init}_{shn}$ and $\mathcal{G}^{init}_{min}$, via channel capacity, Shannon entropy and Min entropy, respectively. If our checker terminates and all synthesized patches are applied at runtime, then our framework guarantees that the channel capacity (respectively, Shannon entropy and Min entropy) will remain $\mathcal{G}^{init}_{cap}$ (respectively, $\mathcal{G}^{init}_{shn}$ and $\mathcal{G}^{init}_{min}$) even in the presence of attacks captured via $\mathcal{O}_{time}$ and $\mathcal{O}_{trace}$.*

PROOF. Consider the generic attack model $\mathcal{O} : \{h, m\}^* \rightarrow \mathbb{X}$, mapping each execution trace to an element in the countable set $\mathbb{X}$. We assume a victim program $\mathcal{P}$ that exhibits a set of execution traces $TR \subseteq \{h, m\}^*$. From Equation 29, we know that $|\mathcal{O}(TR)|$ decreases with each round of patch synthesis. Given that our checker terminates, we obtain the program $\mathcal{P}$, together with a set of synthesized patches that are applied when $\mathcal{P}$ executes. Assume $TR^f \in \{h, m\}^*$ is the set of execution traces obtained from $\mathcal{P}$ when all patches are systematically applied. Clearly, $\left|\mathcal{O}\left(TR^f\right)\right| = 1$.

The channel capacity of $\mathcal{P}$, upon the termination of our checker, is $\log\left(\left|\mathcal{O}\left(TR^f\right)\right|\right) = \log 1 = 0$. This concludes that the channel capacity does not change even in the presence of attacks $\mathcal{O}_{time}$ and $\mathcal{O}_{trace}$.

For a given distribution λ of sensitive input $\mathcal{K}$, Shannon entropy $\mathcal{G}^{init}_{shn}$ is computed as follows:

$$\mathcal{G}^{init}_{shn}(\lambda) = -\sum_{\mathcal{K}\in\mathbb{K}} \lambda(\mathcal{K}) \log_2 \lambda(\mathcal{K}) \tag{30}$$

where $\mathbb{K}$ captures the domain of sensitive input $\mathcal{K}$. For a given equivalence class of observation $o \in \mathcal{O}\left(TR^f\right)$, the remaining uncertainty is computed as follows:

$$\mathcal{G}^{final}_{shn}(\lambda_o) = -\sum_{\mathcal{K}\in\mathbb{K}} \lambda_o(\mathcal{K}) \log_2 \lambda_o(\mathcal{K}) \tag{31}$$

$\lambda_o(\mathcal{K})$ captures the probability that the sensitive input is $\mathcal{K}$, given the observation o is made by the attacker. Finally, to evaluate the remaining uncertainty of the patched program version, $\mathcal{G}^{final}_{shn}(\lambda_o)$ is averaged over all equivalence class of observations as follows:

$$\mathcal{G}^{final}_{shn}\left(\lambda_{\mathcal{O}\left(TR^f\right)}\right) = \sum_{o\in\mathcal{O}\left(TR^f\right)} pr(o)\, \mathcal{G}^{final}_{shn}(\lambda_o) \tag{32}$$

where $pr(o)$ captures the probability of the observation $o \in \mathcal{O}\left(TR^f\right)$. However, we have $\left|\mathcal{O}\left(TR^f\right)\right| = 1$. Hence, for any $o \in \mathcal{O}\left(TR^f\right)$, we get $pr(o) = 1$ and $\lambda_o(\mathcal{K}) = \lambda(\mathcal{K})$. Plugging these observations into Equation 32 and Equation 31, we get the following:

$$\boxed{\begin{aligned}\mathcal{G}^{final}_{shn}\left(\lambda_{\mathcal{O}\left(TR^f\right)}\right) &= \sum_{o\in\mathcal{O}\left(TR^f\right)} \mathcal{G}^{final}_{shn}(\lambda_o) \\ &= -\sum_{\mathcal{K}\in\mathbb{K}} \lambda(\mathcal{K}) \log_2 \lambda(\mathcal{K}) \\ &= \mathcal{G}^{init}_{shn}(\lambda)\end{aligned}} \tag{33}$$

Finally, for a given distribution λ of sensitive input $\mathcal{K}$, the min entropy $\mathcal{G}^{init}_{min}$ is computed as follows:

$$\mathcal{G}^{init}_{min}(\lambda) = -\log_2 \max_{\mathcal{K}\in\mathbb{K}} \lambda(\mathcal{K}) \tag{34}$$

Therefore, min entropy captures the best strategy of an attacker, that is, to choose the most probable secret.

Similar to Shannon entropy, for a given equivalence class of observation $o \in \mathcal{O}\left(TR^f\right)$, the remaining uncertainty is computed as follows:

$$\mathcal{G}^{init}_{min}(\lambda_o) = -\log_2 \max_{\mathcal{K}\in\mathbb{K}} \lambda_o(\mathcal{K}) \tag{35}$$

$\lambda_o(\mathcal{K})$ captures the probability that the sensitive input is $\mathcal{K}$, given the observation o is made by the attacker.

Finally, we obtain the min entropy of the patched program version via the following relation:

$$\mathcal{G}^{final}_{min}\left(\lambda_{\mathcal{O}\left(TR^f\right)}\right) = -\log_2 \sum_{o\in\mathcal{O}\left(TR^f\right)} pr(o) \max_{\mathcal{K}\in\mathbb{K}} \lambda_o(\mathcal{K}) \tag{36}$$

Since $pr(o) = 1$ and $\lambda_o(\mathcal{K}) = \lambda(\mathcal{K})$ for any $o \in \mathcal{O}\left(TR^f\right)$, we get the following from Equation 36 and Equation 35:

$$\boxed{\begin{aligned}\mathcal{G}^{final}_{min}\left(\lambda_{\mathcal{O}\left(TR^f\right)}\right) &= -\log_2 \sum_{o\in\mathcal{O}\left(TR^f\right)} \max_{\mathcal{K}\in\mathbb{K}} \lambda_o(\mathcal{K}) \\ &= -\log_2 \max_{\mathcal{K}\in\mathbb{K}} \lambda(\mathcal{K}) \\ &= \mathcal{G}^{init}_{min}(\lambda)\end{aligned}} \tag{37}$$

Equation 33 and Equation 37 conclude this proof.

□

Modeling LRU and FIFO cache semantics

To formulate the conditions for conflict misses in set-associative caches, it is necessary to understand the notion of cache conflict. We use the following definition of cache conflict to formulate $\Gamma(r_i)$:

DEFINITION 1. ***(Cache conflict)*** r_j generates a cache conflict to r_i if and only if $1 \leq j < i$, $\sigma(r_j) \neq \sigma(r_i)$ and the execution of r_j can change the relative position of $\sigma(r_i)$ within the $set(r_i)$-state immediately before instruction r_i.

Recall that $\sigma(r_i)$ captures the memory block accessed at r_i and $set(r_i)$ captures the cache set accessed by r_i. The state of a cache set is an ordered $\mathcal{A}$-tuple – capturing the relative positions of all memory blocks within the respective cache set. For instance, $\langle m_1, m_2 \rangle$ captures the state of a two-associative cache set. The rightmost memory block (*i.e.* m_2) captures the first memory block to be evicted from the cache set if a block $m \notin \{m_1, m_2\}$ is accessed and mapped to the same cache set.

Challenges with LRU policy. To illustrate the unique challenges related to set-associative caches, let us consider the following sequence of memory accesses in a two-way associative cache and with LRU replacement policy: $(r_1 : m_1) \rightarrow (r_2 : m_2) \rightarrow (r_3 : m_2) \rightarrow (r_4 : m_1)$. We assume both m_1 and m_2 are mapped to the same cache set. If the cache is empty before r_1, r_4 will still incur a *cache hit*. This is because, r_4 suffers cache conflict only once, from the memory block m_2. To incorporate the aforementioned phenomenon into our cache semantics, we only count cache conflicts from the closest access to a given memory block. Therefore, in our example, we count cache conflicts to r_4 from r_3 and discard the cache conflict from r_2. Formally, we introduce the following additional condition for instruction r_j to inflict a cache conflict to instruction r_i.

$\phi_{\mathbf{ji}}^{\mathbf{eqv,lru}}$: No instruction between r_j and r_i accesses the same memory block as r_j. This is to ensure that r_j is the closest to r_i in terms of accessing the memory block $\sigma(r_j)$. We capture $\phi_{ji}^{eqv,lru}$ formally as follows:

$$\phi_{ji}^{eqv,lru} \equiv \bigwedge_{j<k<i} \left(\rho_{jk}^{tag} \vee \neg \rho_{jk}^{set} \vee \neg guard_k \right) \tag{38}$$

Hence, r_j inflicts a unique cache conflict to r_i only if $\phi_{ji}^{eqv,lru}$, $\phi_{ji}^{cnf,lru} \equiv \phi_{ji}^{cnf,dir}$, $\phi_{ji}^{rel,lru} \equiv \phi_{ji}^{rel,dir}$ are all satisfiable.

Challenges with FIFO policy. Unlike LRU replacement policy, the cache state does not change for a cache hit in FIFO replacement policy. For example, consider the following sequence of memory accesses in a two-way associative FIFO cache: $(r_1 : m_1) \rightarrow (r_2 : m_2) \rightarrow (r_3 : m_1) \rightarrow (r_4 : m_1)$. Let us assume m_1, m_2 map to the same cache set and the cache is empty before r_1. In this example, r_2 generates a cache conflict to r_4 even though m_1 is accessed between r_2 and r_4. This is because r_3 is a cache hit and it does not change cache states.

In general, to formulate $\Gamma(r_i)$, we need to know whether any instruction r_j, prior to r_i, was a cache miss. This, in turn, is captured via $\Gamma(r_j)$. Concretely, r_j generates a unique cache conflict to r_i if all the following conditions are satisfied.

$\phi_{\mathbf{ji}}^{\mathbf{cnf,fifo}}$: If r_j accesses the same cache set as r_i, but accesses a different cache-tag as compared to r_i and r_j suffers a cache miss. This is formalized as follows:

$$\phi_{ji}^{cnf,fifo} \equiv \rho_{ji}^{tag} \wedge \rho_{ji}^{set} \wedge \Gamma(r_j) \tag{39}$$

$\phi_{\mathbf{ji}}^{\mathbf{rel,fifo}}$: No cache miss between r_j and r_i access the same memory block as r_i. $\phi_{ji}^{rel,fifo}$ ensures that the relative position of the memory block $\sigma(r_i)$ within $set(r_i)$ was not reset between r_j and r_i. $\phi_{ji}^{rel,fifo}$ is formalized as follows:

$$\phi_{ji}^{rel,fifo} \equiv \bigwedge_{j<k<i} \left(\rho_{ki}^{tag} \vee \neg \rho_{ki}^{set} \vee \neg guard_k \vee \neg \Gamma(r_k) \right) \tag{40}$$

$\phi_{\mathbf{ji}}^{\mathbf{eqv,fifo}}$: No cache miss between r_j and r_i access the same memory block as r_j. We note that $\phi_{ji}^{eqv,fifo}$ ensures that r_j is the closest cache miss to r_i accessing the memory block $\sigma(r_j)$. This, in turn, ensures that we count the cache conflict from memory block $\sigma(r_j)$ to instruction r_i *only once*. We formulate $\phi_{ji}^{eqv,fifo}$ as follows:

$$\phi_{ji}^{eqv,fifo} \equiv \bigwedge_{j<k<i} \left(\rho_{jk}^{tag} \vee \neg \rho_{jk}^{set} \vee \neg guard_k \vee \neg \Gamma(r_k) \right) \tag{41}$$

Formulating cache conflict in set-associative caches. With the intuition mentioned in the preceding paragraphs, we formalize the unique cache conflict from r_j to r_i via the following logical conditions:

$$\Theta_{j,i}^{+,x} \equiv \left(\phi_{ji}^{cnf,x} \wedge \phi_{ji}^{rel,x} \wedge \phi_{ji}^{eqv,x} \wedge guard_j \right) \Rightarrow \left(\eta_{ji} = 1 \right) \tag{42}$$

$$\Theta_{j,i}^{-,x} \equiv \left(\neg \phi_{ji}^{cnf,x} \vee \neg \phi_{ji}^{rel,x} \vee \neg \phi_{ji}^{eqv,x} \vee \neg guard_j \right) \Rightarrow \left(\eta_{ji} = 0 \right) \tag{43}$$

where $x = \{lru, fifo\}$. Concretely, η_{ji} is set to 1 if r_j creates a unique cache conflict to r_i and η_{ji} is set to 0 otherwise.

Computing $\Gamma(r_i)$ for Set-associative Caches. To formulate $\Gamma(r_i)$ for set-associative caches, we need to check whether the number of unique cache conflicts to r_i exceeds the associativity ($\mathcal{A}$) of the cache. Based on this intuition, we formalize $\Gamma(r_i)$ for set-associative caches as follows:

$$\boxed{\Gamma(r_i) \equiv guard_i \wedge \left(\Theta_i^{cold} \vee \left(\left(\sum_{j \in [1,i)} \eta_{ji} \right) \geq \mathcal{A} \right) \right)} \tag{44}$$

We note that $\sum_{j \in [1,i)} \eta_{ji}$ accurately counts the number of unique cache conflicts to the instruction r_i (*cf.* Equation 42-Equation 43). Hence, the condition $\left(\sum_{j \in [1,i)} \eta_{ji} \geq \mathcal{A} \right)$ precisely captures whether $\sigma(r_i)$ is replaced from the cache before r_i is executed. If r_i does not suffer a cold miss and $\left(\sum_{j \in [1,i)} \eta_{ji} < \mathcal{A} \right)$, then r_i will be a cache hit when executed, as captured by the condition $\neg \Gamma(r_i)$.

Detailed Runtime Monitoring

Algorithm 3 outlines the overall process. The procedure MONITORING takes the following inputs:

- Ψ: A symbolic representation of program and cache semantics with the current level of abstraction,
- O and φ: The model of the attacker (O) and a property φ initially capturing cache side-channel freedom w.r.t. O,

Algorithm 3 Monitor extraction and instrumentation

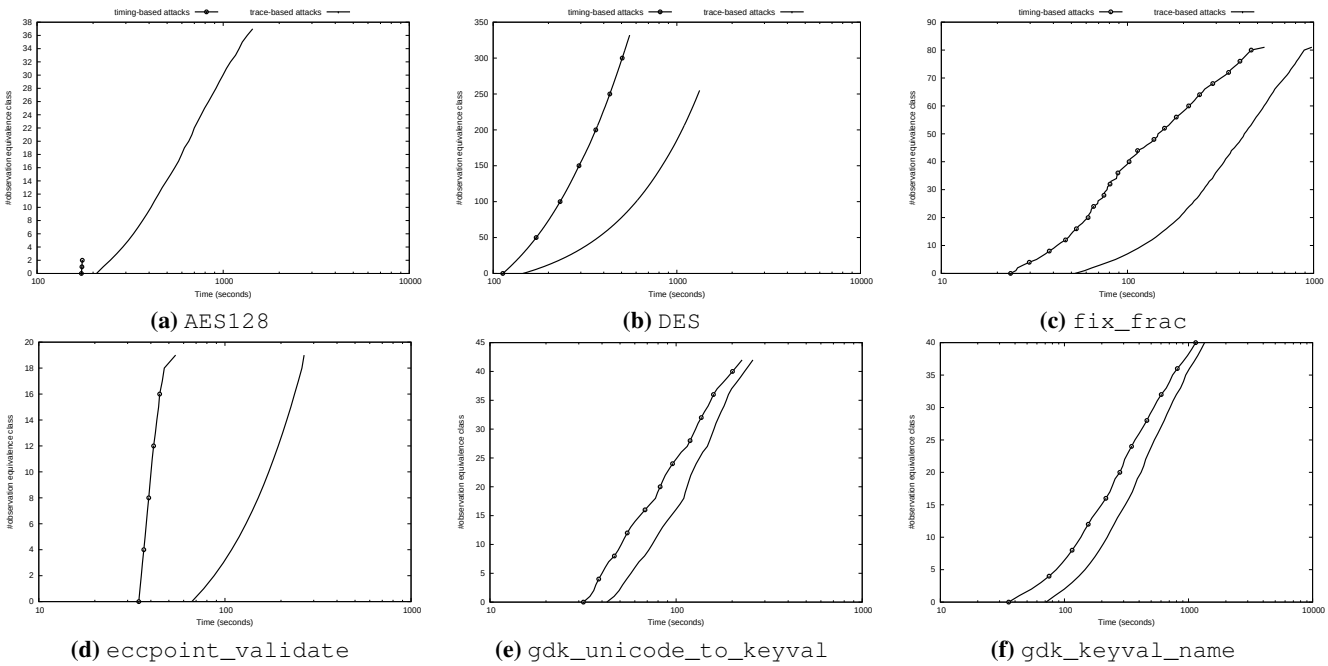
```
1:  procedure MONITORING(Ψ, O, φ, Pred, Pred_cur, Γ)
2:      /* If φ captures side-channel freedom, then trace is
3:      any of the two traces constituting the counterexample */
4:      (res, trace) := VERIFY(Ψ, φ)
5:      while (res=false) ∧ (trace ≠ spurious) do
6:          /* Extract observation from trace */
7:          o := GETOBSERVATION(trace)
8:          /* Extract monitor from trace */
9:          v_o := v := EXTRACTMONITOR(trace)
10:         /* Refine Ψ to find unique counterexamples */
11:         Ψ := Ψ ∧ ¬v
12:         /* Refine φ to find all traces exhibiting o */
13:         φ := REFINEOBJECTIVE(φ, o)
14:         /* Check the unsatisfiability of Ψ ∧ ¬φ */
15:         (res′, trace′) := VERIFY(Ψ, φ)
16:         while (res′=false) do
17:             if (trace′ ≠ spurious) then
18:                 v := EXTRACTMONITOR(trace′)
19:                 /* Combine monitors with observation o */
20:                 v_o := v_o ∨ v
21:                 /* Refine Ψ for unique counterexamples */
22:                 Ψ := Ψ ∧ ¬v
23:                 /* Check the unsatisfiability of Ψ ∧ ¬φ */
24:                 (res′, trace′) := VERIFY(Ψ, φ)
25:             else
26:                 /* Refine abstraction to repeat verification */
27:                 ABSREFINE(Ψ, Pred, Pred_cur, trace′, Γ)
28:                 (res′, trace′) := VERIFY(Ψ, φ)
29:             end if
30:         end while
31:         Let Ω holds the set of monitor, observation pairs
32:         Ω ∪:= {⟨v_o, o⟩}
33:         /* Instrument patches for monitor v_o */
34:         INSTRUMENTPATCH(Ω, O)
35:         /* Refine objective to find new observations */
36:         φ := ¬φ
37:         /* Check the unsatisfiability of Ψ ∧ ¬φ */
38:         (res, trace) := VERIFY(Ψ, φ)
39:     end while
40:     /* Program is still not side-channel free */
41:     /* Refine abstraction to repeat verification loop */
42:     if (res=false) then
43:         ABSREFINE(Ψ, Pred, Pred_cur, trace, Γ)
44:         MONITORING(Ψ, O, φ, Pred, Pred_cur, Γ)
45:     end if
46: end procedure
47: procedure ABSREFINE(Ψ, Pred, Pred_cur, trace, Γ)
48:     /* Extract unsatisfiable core */
49:     U := UNSATCORE(trace, Γ)
50:     /* Refine abstractions (see Section 4) */
51:     Pred_cur := REFINE(Pred_cur, U, Pred)
52:     /* Rewrite Ψ with the refined abstraction */
53:     REWRITE(Ψ, Pred_cur)
54: end procedure
```

- *Pred* and $Pred_{cur}$: Cache semantics related predicates (*Pred*) and the current level of abstraction ($Pred_{cur}$), and
- Γ: Symbolic conditions to determine the cache behaviour.

Additional Experimental Results

Sensitivity w.r.t. cache. Figure 4 outlines the evaluation for routines that violate side-channel freedom and for attack model O_{time}. Nevertheless, the conclusion holds for all routines and attack models. Figure 4 captures the number of equivalence classes explored (hence, the number of patches generated *cf.* Algorithm 3) with respect to time. We make the following crucial observations from Figure 4. Firstly, the scalability of our checker is stable across a variety of cache configurations. This is because we encode cache semantics within a program via symbolic constraints on cache conflict. The size of these constraints depends on the number of memory-related instructions, but its size is not heavily influenced by the size of the cache. Secondly, the number of equivalence classes of observations does not vary significantly across cache configurations. Indeed, the number of equivalence classes may even increase (hence, increased channel capacity) with a bigger cache size (*e.g.* in `DES` and `AES`). However, for all cache configurations, CACHEFIX generated all the patches that need to be applied for making the respective programs cache side-channel free.

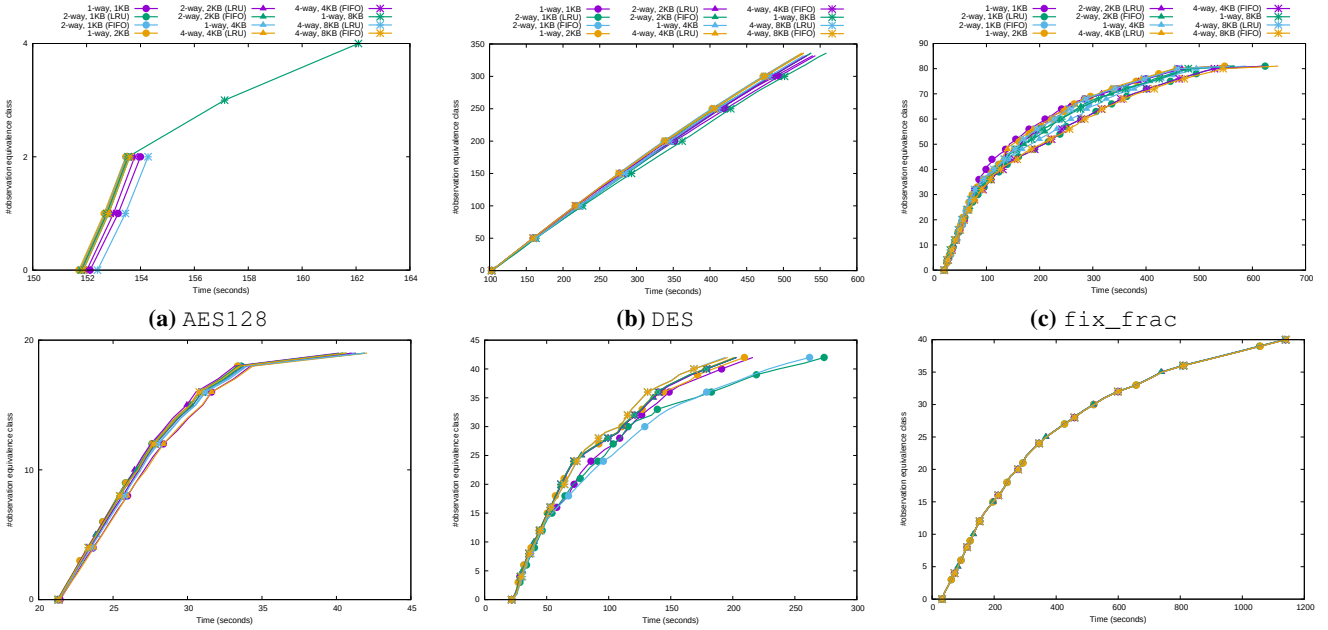
The scalability of CacheFix *is stable across a variety of cache configurations. Moreover, in all cache configurations,* CacheFix *generated all required patches to ensure the cache side-channel freedom w.r.t.* O_{time}.



(a) `AES128` **(b)** `DES` **(c)** `fix_frac`

(d) `eccpoint_validate` **(e)** `gdk_unicode_to_keyval` **(f)** `gdk_keyval_name`

Figure 3: Overhead of counterexample exploration and patch synthesis



(a) `AES128` **(b)** `DES` **(c)** `fix_frac`

(d) `eccpoint_validate` **(e)** `gdk_unicode_to_keyval` **(f)** `gdk_keyval_name`

Figure 4: CACHEFIX sensitivity w.r.t. cache