Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels

Timos Antonopoulos, Paul Gazzillo, Michael Hicks[†], Eric Koskinen, Tachio Terauchi[‡], Shiyi Wei[†]

Yale University, USA [†] University of Maryland, USA [‡] JAIST, Japan

Abstract

We present a novel approach to proving the absence of timing channels. The idea is to partition the program's execution traces in such a way that each partition component is checked for timing attack resilience by a time complexity analysis and that per-component resilience implies the resilience of the whole program. We construct a partition by splitting the program traces at secret-independent branches. This ensures that any pair of traces with the same public input has a component containing both traces. Crucially, the per-component checks can be normal *safety properties* expressed in terms of a single execution. Our approach is thus in contrast to prior approaches, such as *self-composition*, that aim to reason about multiple ($k \geq 2$) executions at once.

We formalize the above as an approach called *quotient partitioning*, generalized to any k-safety property, and prove it to be sound. A key feature of our approach is a demand-driven partitioning strategy that uses a regex-like notion called *trails* to identify sets of execution traces, particularly those influenced by tainted (or secret) data. We have applied our technique in a prototype implementation tool called *Blazer*, based on WALA, PPL, and the brics automaton library. We have proved timing-channel freedom of (or synthesized an attack specification for) 24 programs written in Java bytecode, including 6 classic examples from the literature and 6 examples extracted from the DARPA STAC challenge problems.

CCS Concepts • **Security and privacy** → **Logic and verification**; • **Theory of computation** → **Program reasoning**; *Logic and verification*; • **Software and its engineering** → *Formal methods*

Keywords Timing Attacks, Blazer, Verification, Subtrails, Decomposition

1. Introduction

A program has a *timing channel* if variations in its running time can reveal information about secret inputs. For example, a password check whose running time depends on how much of the guess matches the real (secret) password potentially has a timing channel. That a program is free of timing channels is a k-*safety property*. Such a property involves k (terminating) runs of a program and, consequently, validation requires establishing relationships between k different execution traces. For timing channels, we need $k = 2$ runs: any pair of executions whose computations use the same public inputs (but may differ on their secret inputs) should nevertheless have similar running times.

We are interested in verifying that programs are free of timing channels. It seems appealing to leverage the success of abstract interpretation. Abstract interpretation-based tools enjoy rigorous guarantees and provide formal proofs of various safety and liveness properties. They also are efficient. Implementations such as ASTRÉE and INTERPROC are able to validate properties of large C programs.

Abstract interpretation-based techniques focus on single executions, but to prove k-safety properties we need to relate multiple executions. A clever way to do this is to employ self-composition [7, 8]: To reason about k runs of a program, we can concatenate k copies of it (with variables suitably renamed) and then assert a property that relates variables in different copies. For our timing channel property, we could concatenate the program with itself, require that public inputs to both copies be the same, and then assert that execution counters inserted for each copy are (approximately) equal at the conclusion, despite allowed variation of secret inputs. Self-composition can be expensive due to the explosion of the cross-product state space. As such, several works have looked to improve the basic idea by exploiting structural similarities in the program (e.g., using *interleaving composition* [27, 36, 37]). Other works applied similar ideas to improve relational verifiers directly [5, 11, 31, 34]. As discussed Section 7, these approaches (and others) nevertheless rely, at least implicitly, on making k copies of the program, which means that invariants are split across the product program. The result can be either poor performance

or key information being lost due to abstraction during fixpoint computation.

With a focus on verifying the absence of timing channels, we depart from the composition-based strategies and instead establish a novel *decomposition* methodology. Our key insight, sketched in Section 2, is that rather than prove a relational property about all pairs of execution traces, we can prove a non-relational property about each trace in a certain trace partition, computed iteratively.

Formally, let C be a program of interest, and $[\![C]\!]$ be all of its possible execution traces. Our method aims to produce a partition $\mathfrak{T} = T_1, ..., T_n$ (for some $n \geq 1$) of $[\![C]\!]$ which has the following properties:

1. For every pair of traces $\{\pi_1, \pi_2\} \subseteq [\![C]\!]$, if $in(\pi_1)[\mathsf{low}] = in(\pi_2)[\mathsf{low}]$ then there exists some T_i such that $\{\pi_1, \pi_2\} \in T_i$.[1] That is, if two execution traces have the same *low*—meaning non-*secret*—inputs, then they must both be part of some partition element.
2. For each T_i and $\pi \in T_i$ there exists a function f such that $time(\pi) = f(in(\pi)[\mathsf{low}]) \pm c$; *i.e.*, the running time of each trace is a function of the low input, plus or minus some constant c (which defines the observational limits of the attacker). Importantly, the running time should *not* be a function of the secret inputs.

With such a partition we establish the *timing channel freedom* property Φ_{tcf} we desire, which is $\forall \pi_1, \pi_2 \in [\![C]\!]$,

$$in(\pi_1)[\mathsf{low}] = in(\pi_2)[\mathsf{low}] \Rightarrow time(\pi_1) \approx time(\pi_2)$$

Each trace in the same partition component T_i has the same running time function f over low variables (only), and traces in different partitions have unequal low inputs and thus satisfy Φ_{tcf} vacuously.

In Section 3, we make this point rigorous by formally developing a decomposition strategy we call *k-quotient partitioning* for proving k-safety properties Φ, showing that our proof of Φ_{tcf} is an instance of it. k-quotient partitioning has two components: (1) a k-ary *quotient formula* ψ such that each set of k traces that satisfies ψ appears in the same partition element; and (2) a family of k-ary *partition properties* P_i such that any set of k traces in a partition element T_i satisfy P_i, and all P_i together jointly imply the global property Φ. For timing channels, we have $k = 2$, the quotient formula is that traces' low inputs are equal, and the partition property is that the running time of each partition element's trace is a function f of its low inputs. Section 3 briefly describes other k-safety properties whose proof may be amenable to k-quotient partitioning.

A key challenge of k-quotient partitioning is arriving at the partition $\mathfrak{T}$. Our approach to proving the absence of timing channels synthesizes $\mathfrak{T}$ iteratively, along with the corresponding per-component running times P_i. We do this using an abstraction called *trails*. A trail t is an annotated regular expression (regex) over the edges of the control-flow graph, and specifies a subset of the program's execution traces (i.e., those that follow the trail). We partition execution traces using trails annotated by the influence of secret or public inputs ("taint"). In particular, branching expressions in a trail (union or Kleene star constructors) are annotated as to whether the branch depends on inputs that are tainted. Starting with the most general trail, which captures all executions, we attempt to prove a tight lower and upper bound on the running time of traces described by the trail [16–18] by matching transition relations with a database of lemmas [17]. If we cannot, we break the trail into smaller trails at tainted branching expressions; *e.g.*, putting all executions that take the **true** branch into one trail and all that take the **false** branch into another. Since we split at branches that are taint-influenced, we ensure that execution pairs involving equal public inputs will be put in the same partition component. The trail abstraction and our partitioning algorithm is described in detail in Section 4.

Section 5 describes the implementation of our approach in a tool called *Blazer*. We equip a standard abstract interpreter with the ability to consult an oracle (the synthesized trails) to decide which CFG arcs to follow, thus deriving partition-specific invariants. These invariants underlie the per-trail running time analysis. If *Blazer* cannot prove that a partition T_i has tight bounds on running times, and further taint-based sub-partitions are not possible, it attempts to synthesize possible attacks. In particular, it generates sub-partitions and running times based on secret information; if a difference in secret values results in observable differences in running time, then there is a possible attack.

Section 6 presents an evaluation of *Blazer* on a collection of 25 benchmarks, including 12 tricky hand-crafted benchmarks, 7 programs from the literature [15, 22, 29], and 6 fragments of the DARPA STAC challenge problems [35]. We find that *Blazer* is able to prove the absence of timing channels when the program is safe or else synthesize an attack specification in all but two cases.

In summary, this paper contributes:

1. A novel program analysis technique for proving the absence of timing channels that decomposes the problem into one of proving simpler properties of subproblems rather than proving the relational (2-safety) property directly, e.g., by self-composition (Sec. 2).
2. A proof of soundness of a general decomposition approach for proving k-safety properties called k-quotient partitioning, of which our timing channel approach is one instance (Sec. 3).
3. A novel symbolic representation of partitions, *trails*, and an algorithm for constructing new partitions (Sec. 4).

[1] Technically, the $T_1, ..., T_n$ need not be disjoint, *i.e.*, the same trace π may appear in several T_i. But for gaining intuition this detail is unimportant.

4. An implementation of our approach in the tool *Blazer* discovering timing channels in Java programs, or proving their absence (Sec. 5).[2]
5. An evaluation on 25 benchmark programs, including examples from the STAC challenge problems and the cryptography literature [15, 22, 29] (Sec. 6).

2. Overview

In this section, we give a tour of our work. We begin with simple examples to introduce the key idea of using a partition of executions to prove timing channel freedom. We then describe our algorithm in more detail, explaining how partitioning is interleaved with running time computation. Finally, we explain how the same basic approach can be used to synthesize a possible attack when timing channel freedom cannot be proved.

2.1 Decomposition: The Basics

Consider the following example program.

Example 1.

```
1   void foo(int high, uint low) {
2     if (high == 0) {
3       i = 0;
4       while(i < low) i++;
5     }
6     else {
7       i = low;
8       while(i > 0) i−−;
9     }
10  }
```

This program takes a secret input high, and an attacker-controlled (tainted) input, low. A program is said to have a *timing channel* if the attacker, assumed to know the program code, can infer information about the high by observing the program's running time (perhaps iteratively varying the input low). The execution of this particular program branches on secret variable high so we may wonder whether there is a timing channel. Proving there is not one requires, in principle, that we relate *all pairs of execution traces*. Doing so directly (*e.g.* by constructing a self-composition [7, 36] or a relational program analysis [5, 9, 11, 34, 40]) can magnify the overall state space to consider.

We show that we can instead prove that all executions (later: execution partitions) *share the same property*. For timing channels, we want to prove that each execution of the above program has a running time that is a function of (only) low. In this case we can be more specific: the running time is *linear* in low (for some fixed linear function). We might write this as a property $P^{\text{lin}}_{\mathsf{low}}$, and then write:

$$\forall \pi \in [\![C]\!].\ P^{\text{lin}}_{\mathsf{low}}(\pi)$$

[2] While *Blazer* does not yet support recursive functions, our theoretical results still apply; we plan to explore an annotation-based strategy.

where $[\![C]\!]$ is the set of all execution traces of the program C. An obvious consequence of this is that *every pair* of executions π_1, π_2 share $P^{\text{lin}}_{\mathsf{low}}$. As such, it is clear that there can be no timing channel.

The key idea of our approach is to break down the executions of the program into various cases, depending on high-independent branching, and in each case discover a running time property P that describes all traces in that case. To do this, we symbolically (and automatically) discover a partitioning of the execution traces $\mathfrak{T} = T_1, ..., T_n$ such that $[\![C]\!] = \bigcup_{i\in[1,n]} T_i$ so that we can find some P_i to characterize the running time for all $\pi \in T_i$. As long as each P_i is independently acceptable (*i.e.*, running time does not depend (much) on high), then the overall program satisfies the desired property. In Example 1, we only needed one partition component. Here is another example.

Example 2.

```
1   void bar(int high, int low) {
2     int i;
3     if (low > 0) { // O(2*low)
4       i = 0;
5       while(i<low) i++;
6       while(i>0) i−−;
7     } else { // O(1)
8       if (high == 0) { i = 5; } else { i = 0; i++; }
9     }
10  }
```

In this program, not all executions have the same symbolic running time. If low is positive, the execution will be linear in low, otherwise it will be either one instruction or two, depending on the value of high. To discover this, we can form a partition of the execution traces as follows:

$$T_> \triangleq \{\pi \mid in(\pi)[\mathsf{low}] > 0\} \quad \text{and} \quad T_\leq \triangleq \{\pi \mid in(\pi)[\mathsf{low}] \leq 0\}$$

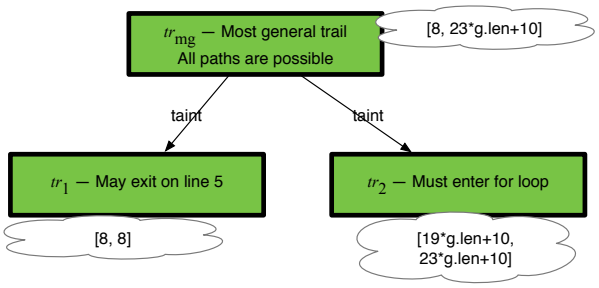
where $[\![C]\!] \subseteq T_> \cup T_\leq$ and $in(\pi)[\mathsf{low}]$ means the value of the low input variable of trace π. For now, let us assume this partition is given to us (we describe how we get the partition shortly). With some help (discussed below), we can coerce an off-the-shelf abstract interpreter to now perform two analyses proving, respectively, that:

$$\forall \pi \in T_>.\ P^{\text{lin}}_{\mathsf{low}}(\pi) \quad \text{and} \quad \forall \pi \in T_\leq.\ P^{\text{const}}_{\text{c}}(\pi).$$

Here, property $P^{\text{const}}_{\text{c}}(\pi)$ means that the running time of the trace is within some (fixed) small attacker-unobservable bound (say, ε) of the constant c. For this example, $c = 1$ with $\varepsilon = 1$.

These two proofs establish relationships between any two *copartitional* traces (two traces in $T_>$ or two traces in $T_\leq$). But what about some $\pi_0 \in T_>$ and some $\pi_1 \in T_\leq$? These two traces have different symbolic running times. However, notice that the path condition (whether low is above 0) depends only on variable low and not on secret variable high. Consequently, we can immediately conclude that, although

```
1  loginSafe(String username, byte[] guess) {
2    boolean dummy, matches = true;
3    byte[] user_pw = retrievePassword(username);
4    if(user_pw == null)
5      return false;
6    for(int i = 0; i < guess.length; i++) {
7      if (i < user_pw.length) {
8        if(guess[i] != user_pw[i]) matches = false;
9        else dummy = true;
10     } else {
11       dummy = true; matches = false;
12     }
13   }
14   return matches; }
```




```
1  loginBad(String username, byte[] guess) {
2    byte[] user_pw = retrievePassword(username);
3    if(user_pw == null)
4      return false;
5    for(int i = 0; i < guess.length; i++) {
6      if (i < user_pw.length) {
7        if(guess[i] != user_pw[i])
8          return false;
9      } else {
10       return false;
11     }
12   }
13   return true; }
```

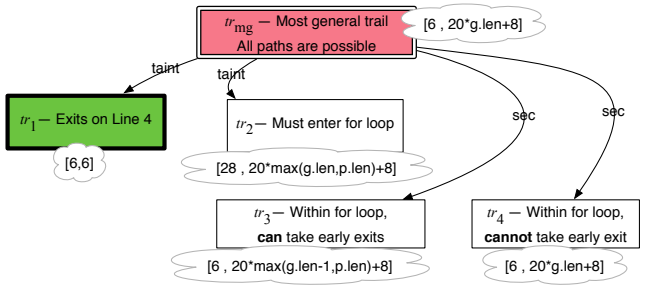


Figure 1. Two versions of a program that validates a password (left column) and the corresponding output of our tool *Blazer* (right column). Each rectangle represents a trail, and each downarrow from a trail represents a subtrail; the taint and sec annotations indicate on what sort of data the subtrail was created. The balloons contain ranges $[x, y]$ that indicate the lower/upper bound on the trail's running time in terms of number of instructions (g.len is short for guess.length and p.len is short for user_pw.length). Bolded (green) nodes indicate areas where the program is safe and double-line (red) nodes indicate an attack specification.

traces from two different partition components have different running times, these differences cannot be correlated with high's value.

2.2 Synthesizing Partitions with Trails

Our trace partition in Example 2 considered different input values: those for which low > 0, or not. This distinction corresponds to whether we branch at line 3, or not. Our algorithm likewise develops a partition by considering sets of paths, particularly those that take a branch one way vs. the other. Paths are specified using annotated regular expressions tr that we call *trails*, which for the purposes of this example have the following grammar:

$$tr ::= ij \;\|\; i{\downarrow} \;\|\; tr_1 \cdot tr_2 \;\|\; tr_1 \,|_\alpha\, tr_2 \;\|\; tr_\alpha^*$$

Trails are defined over edges ij in a control-flow graph, where i and j represent CFG blocks in the program. We also permit edge $i{\downarrow}$, meaning a block i that subsequently moves to the exit block. We use $\cdot$ for sequential composition (often dropped when clear from context), vertical bar for branching, and star for looping. Both branching and looping regex operators are annotated with $\alpha \in \{l,h,(l,h)\}$. Symbol l is used to mean low, h to mean high, or l,h to mean both. Here is an example:[3]

$$23 \cdot (34 \cdot 45 \cdot 5_l^* \cdots) \,|_l\, (38 \cdots)$$

This represents executions of Example 2. These executions start on line 2 and proceed to line 3, characterized by edge 23. Then they follow the branch at line 3, which depends on low input, as indicated by the annotation l. According to the lhs of the branch, this trail considers executions that go from line 3 to 4 and then 4 to 5 and then 0 or more times through line 5, due to the low-dependent loop, etc. According to the

[3] For this example we refer to line numbers, rather than CFG blocks (as used by our actual algorithm), for clarity.

rhs of the branch $|_l$, the trail also considers executions that go from 3 to 8 (etc.).

The trail-based partition is generated iteratively. We start with a trail that characterizes all possible executions of the program; we call this the *most general trail*, tr_{mg}. We attempt to compute the running time of all paths characterized by this trail. If the running time is primarily a function of low input, with constant-bounded effect from high, then we are done. If this is not possible, we further break down the trail at low-dependent (only) branching points. For Example 2, this results in a trail describing all paths for all true paths through line 3, and another for all false paths.

Ultimately, the final partition is a collection of trails $tr_1, ..., tr_n$ such that:

1. The union of all of the trails' languages covers the language of tr_{mg}. That is, $\bigcup_{i\in[1,n]} L(tr_i) \supseteq L(tr_{mg})$.
2. Trails correlate to the branching decisions that depend only on low security variables.
3. Each trail tr_i is such that every execution's running time can be described by a single function P_i, *e.g.* $P_{\mathsf{low}}^{\mathsf{lin}}$. Hence it can be proved that $\forall \pi \in L(tr_i) \cap [\![C]\!]$, that $\pi \models P_i$.

Thanks to the formal guarantees of our decomposition (Section 3), this suffices to entail that the overall program is free of timing channels.

More elaborate example. Consider the program loginSafe at the top of Figure 1. This function looks up the given username (a non-secret) and if the user is known, checks that the given guess matches the user's password. We return true if so, and false otherwise.[4]

To the right of the code is a visual depiction of the a tree of trail specifications. The top box represents the most general trail, which is our starting point. With it, we compute the running time of executions that follow this trail. This is embodied in a component called BOUNDANALYSIS. Technically, to implement BOUNDANALYSIS we equip an off-the-self abstract interpreter with the ability to be restricted to a given trail, leverage the seeding technique [10] to compute transition invariants [30], and match these invariants against a database of complexity bound lemmas [16, 17]. The result of applying BOUNDANALYSIS to tr_{mg} is that it computes a lower bound of 8 and an upper bound of $23 \times \mathsf{g.len} + 10$ (depicted as a cloud), where g.len is shorthand for guess.length.

With the range of running times in hand, we now ask whether the range is *narrow*: that the running time of the given executions is a function of low variables plus up to a maximum constant value c, where c is a limit on the observability of the attacker, *i.e.*, the difference between the longest and shortest running time is c. If we find that the range between a given lower/upper bound is narrow then we know we mark the code as free from timing channels. For tr_{mg}, we find that this bound is *not* narrow. This means we need to partition tr_{mg} according to (only) tainted data, and then try again, for each partition.

We do this using a component called REFINEPARTITION. In this case it splits the most general trail into two based on the branch at line 4. The parent-child relationship in the tree shown in the figure tracks this subtrail relationship. Edges are annotated as to whether the subtrail was chosen based on branching on low data (when trying to prove safety) or based on branching on high data (when trying to synthesize an attack, discussed shortly). For legibility, we have replaced each trail's regular expression with an intuitive description of the trail it specifies.

Now we compute the running times for these two trails separately. The left-hand trail has running time 8—this is the trail that exits on line 4. The right side has a running time that is a direct function of g.len. These two running times individually satisfy our narrowness criterion for timing channel freedom. In particular, the constant one does trivially, and the range on the right side does assuming that $\mathsf{g.len} \leq n$ for some fixed size that does not exceed the power of the attacker (which can be specified); e.g., if $n = 100$, the running time is equivalent to $21 \times \mathsf{g.len} \pm 210$, which is safe assuming the adversary's observational power is bounded by the constant 210. As such, because each one is acceptable, the two together are as well, and the program is sure to be free of timing channels.

2.3 Synthesizing Attacks

Our partitioning strategy is also useful for finding possible attacks if a program is not free of timing channels. To see how this works, consider the bottom of Figure 1. For this example, there is a timing channel (based on a bug in the Tenex password checker [24])—if the program exits on either lines 8 or 10, then the running time reveals the length of the prefix of guess that matches the real password.

When trying to prove this program is timing-channel free, the bound analysis will discover running time bounds similar to the correct example: a lower bound of 6 and an upper bound of $20 \times \mathsf{g.len} + 8$. As such, it will partition again according to the tainted branch on line 3. This time, though, the running times of the two partition trails do not meet our narrowness criterion: while the trail on the left hand side is narrow (due to the exit on line 4), the trail on the right hand side is not narrow: it has a constant lower bound and an upper bound that relies on both g.len and the length of the actual password. Unfortunately, we cannot easily sub-partition the right side further, since branches in the loop also depend on secret information—partitioning is only permitted on low data. As such we have found a potential problem.

At this point, the tool changes gears and attempts to discover a vulnerability. To this end, REFINEPARTITION generates the next two trails (tr_3 and tr_4). These trails differ based on whether or not the shortcut **return** statement can be taken on Line 7 (or 10). For both tr_3 and tr_4, our toolchain computes a lower bound of constant running time, corresponding

[4] We do not consider the presence or non-presence of a username in the database to be secret information, for this example.

to the early exit on line 4. For tr_4, our toolchain computes a *linear* running time of $20 \times \mathsf{g.len} + 8$ for the upper bound. Meanwhile, tr_3 forces the program to return early after entering the loop, either by taking the **return** statement on Line 8 or the **return** statement on Line 10. For tr_3, our toolchain computes a range of running times, different from the ones computed for tr_4. With this, the tool reports a vulnerability: there are two trails (tr_3 and tr_4), the choice between them depends on high data (arcs labeled sec), and yet they have different running times. Therefore, for the same low input there can be two different possible executions that yield different running times as the branching depends on the value of the secret.

We call this output an *attack specification*. Because we are working with a static analysis, the result of our tool is not immediately two concrete traces. However, it provides a specification for two traces that witness the attack. All that remains is to ensure that these traces are feasible by finding justifying inputs. This can be done manually by a programmer or via an under-approximate analysis (*e.g.*, a symbolic execution [29]).

3. Semantic Partitioning

While our practical focus is timing channels, our decomposition strategy works for other k-safety properties, too. In this section we provide a formalization of our general decomposition. The key result is Theorem 3.1 which says that we can conclude a k-safety property of the traces of a program, using non-relational reasoning. Proving timing channel freedom, which is a 2-safety property, is a consequence of this result. The details of our particular algorithm are given in the sections that follow.

3.1 Programs, Traces, and k-Safety Properties

Let π range over execution traces. For a program C, we write $[\![C]\!]$ for the set of all execution traces of C. A k-safety property is a predicate $q(C)$ of the following form:

$$q(C) \triangleq \forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k . \Phi_q(\pi_1, \ldots, \pi_k)$$

where Φ_q is a predicate on k-tuples of execution traces.

We may assume that a program trace contains whatever information needed for the property, such as inputs, outputs, intermediate states, and event times. For example, assuming that π is a sequence of states in the execution, we may refer to the input value of the variable x in the execution trace π by $in(\pi)[x]$.

3.2 Quotient Partition and Quotient Partitionable Properties

A *trace partition* $\mathfrak{T}$ for C is a finite set of non-empty sets of traces $\mathfrak{T} = \{T_1, \ldots, T_n\}$ such that $[\![C]\!] \subseteq \bigcup_{1 \le i \le n} T_i$. Even though we use the name "partition", we do not enforce the T_i's to be pairwise disjoint. Let ψ be a predicate on k-tuples of execution traces. We say that $\mathfrak{T}$ is a *ψ-quotient partition* if the following is satisfied

$$\forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k . \\ \psi(\pi_1, \ldots, \pi_k) \Rightarrow \exists T \in \mathfrak{T} . \textstyle\bigwedge_{1 \le i \le k} \pi_i \in T.$$

Roughly, $\mathfrak{T}$ is a ψ-quotient partition if for each k-tuple of traces satisfying ψ, there is an element $T \in \mathfrak{T}$ to which all the k traces belong.

Example 3. Note that, for relational properties (*i.e.*, when $k \ge 2$), any $true$-quotient partition $\mathfrak{T}$ must have an element $T \in \mathfrak{T}$ such that $[\![C]\!] \subseteq T$. □

Example 4. The partitioning based on low (tainted) inputs described earlier can be formalized as a ψ_{tcf}-quotient partition, where $\psi_{tcf}(\pi_1, \pi_2)$ is the predicate $in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}]$ with $\boldsymbol{\ell}$ being the low (taint) variables (at times, we may also write $in(\pi_1) =_{low} in(\pi_2)$ for the latter predicate). For instance, recall the partition $T_>, T_\le$ described in Section 2 where low is a low variable, $T_>$ contains all traces with low > 0, and $T_\le$ contains all traces with low ≤ 0. It is easy to see that $\mathfrak{T} = \{T_>, T_\le\}$ is a ψ_{tcf}-quotient partition. □

Let q be a k-safety property, as defined above. We say that q is *ψ-quotient partitionable* if

$$\forall C . \forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k . \\ (\psi(\pi_1, \ldots, \pi_k) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k).$$

Note that the condition is equivalent to the following: $\forall C . \forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k . \psi(\pi_1, \ldots, \pi_k) \vee \Phi_q(\pi_1, \ldots, \pi_k)$. In short, to verify or refute a ψ-quotient partitionable property it suffices to only consider the k-tuples of execution traces that satisfy ψ.

Example 5. Trivially, any k-safety property is *true*-quotient partitionable. However, as remarked in Example 3, any such partition must have an element containing all the traces. □

Example 6. The timing channel freedom property is the following 2-safety property:

$$tcf(C) \triangleq \forall \pi_1, \pi_2 \in [\![C]\!]^2 . \\ in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}] \Rightarrow time(\pi_1) \approx time(\pi_2)$$

where $\boldsymbol{\ell}$ are low variables and $time(\cdot) \approx time(\cdot)$ says that the running times of the two executions are indistinguishable. It is easy to see that the property is ψ_{tcf}-quotient partitionable with $\psi_{tcf}(\pi_1, \pi_2) \triangleq in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}]$. □

3.3 Relational Analysis via Non-Relational Analysis

As we shall prove below, a quotient partitionable property can be verified by verifying each quotient partition component individually. A key insight of our approach is that, often, individual partition components become isolated enough to be amenable for verification via a non-relational analysis. Next, we formalize the idea.

A non-relational property is of the form $\forall \pi \in [\![C]\!].P(\pi)$ where P is a predicate on an execution trace. We call such a P a *trace property*.

Consider a k-safety property q again. We say that a trace property P is *relational-by-property-sharing* for q, denoted $\mathsf{RBPS}(P, q)$, if the following holds:

$$\forall \pi_1, \ldots, \pi_k. \bigwedge_{1 \leq i \leq k} P(\pi_i) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)$$

Roughly, $\mathsf{RBPS}(P, q)$ says that checking that P holds for individual execution traces is sufficient for checking that Φ_q holds for the k-tuples of the execution traces. Hence, the condition implies that we can check the relational property Φ_q on the tuples of traces from a partition component by checking the non-relational property P on the component.

Example 7. Recall the timing channel freedom property *tcf* from Example 6. Consider a non-relational complexity analysis which, given T, computes the lower and upper bound of the running times for the execution traces in T, and checks if the bound difference is small enough to be indistinguishable by an attacker. The analysis induces a non-relational property $P_f(\pi)$ where f is a high-independent function over traces and $P_f(\pi)$ is true iff the running time of π is close enough to $f(\pi)$. It is easy to see that $\mathsf{RBPS}(P_f, \mathit{tcf})$. □

When q is ψ-quotient partitionable, it is sufficient to check a relational-by-property-sharing non-relational property on each component of a ψ-quotient partition. More formally, we have the following.

Theorem 3.1 (Soundness)**.** *Suppose that q is ψ-quotient partitionable, and $\mathfrak{T}$ is a ψ-quotient partition for a program C. Then, $q(C)$ holds if for each $T \in \mathfrak{T}$, there exists P such that (i) $\mathsf{RBPS}(P, q)$, and (ii) for each $\pi \in [\![C]\!] \cap T$, $P(\pi)$.*

Proof. Let $q(C) \triangleq \forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k.\Phi_q(\pi_1, \ldots, \pi_k)$. Take $\pi_1, \ldots, \pi_k \in [\![C]\!]^k$ arbitrarily. It suffices to show that $\Phi_q(\pi_1, \ldots, \pi_k)$. Because q is ψ-quotient partitionable, $\Phi_q(\pi_1, \ldots, \pi_k)$ iff $\psi(\pi_1, \ldots, \pi_k) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)$.

Therefore, suppose that $\psi(\pi_1, \ldots, \pi_k)$. It suffices to show that $\Phi_q(\pi_1, \ldots, \pi_k)$. Because $[\![C]\!] = \bigcup \mathfrak{T}$, there must be $T \in \mathfrak{T}$ such that $\pi_1 \in T$. Because $\mathfrak{T}$ is a ψ-quotient partition and $\psi(\pi_1, \ldots, \pi_k)$, it follows that $\pi_i \in T$ for all $1 \leq i \leq k$. Let P be a trace property satisfying conditions (*i*) and (*ii*) for T. Then, we have $\Phi_q(\pi_1, \ldots, \pi_k)$ by (*ii*) and the definition of $\mathsf{RBPS}(P, q)$. □

Example 8. Recall the trivial *true*-quotient partition from Example 3. Recall from Example 5 that the any partition in this case must have an element containing all $[\![C]\!]$. Therefore, in this case, Theorem 3.1 only implies that $q(C)$ holds if P holds for all execution traces of C for some relational-by-property-sharing non-relational property P (wrt. $[\![C]\!]$), which is apparent from the definition. □

Example 9. More generally, we would like to find a strong ψ with which a k-safety property q is quotient partitionable, and obtain a corresponding fine-grained ψ-quotient partition. Such a fine-grained partition improves the analysis efficiency as it divides the analysis problem into smaller subproblems, and, perhaps more importantly, can increase the analysis precision whereby the partition components become isolated enough for the non-relational analysis to return precise results.

For example, for *tcf*, applying the non-relational complexity analysis described in Example 7 on the entire trace set $[\![C]\!]$ is unlikely to be successful since different traces are likely to have widely different run times. By contrast, partitioning the traces via the quotient factor $\psi_{\mathit{tcf}} \triangleq in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}]$ allows dividing the problem so that traces that have different low-security values may be analyzed separately. The non-relational analysis may compute widely different running time bounds for different partition components, but the overall verification may still succeed when the computed bounds are narrow within each individual partition component. □

Theorem 3.1 shows the soundness of our method. In fact, the following result shows that the method is also *complete*. That is, for any k-safety property q and a program C satisfying q, there exist a quotient partition and corresponding non-relational properties that can be used to verify that C satisfies q via the method.

Theorem 3.2 (Completeness)**.** *Suppose $q(C)$. Then, there are ψ and $\mathfrak{T}$ such that q is ψ-quotient partitionable, $\mathfrak{T}$ is a ψ-quotient partition for C, and, for each $T \in \mathfrak{T}$, there exists P such that $\mathsf{RBPS}(P, q)$ and $\forall \pi \in [\![C]\!] \cap T.P(\pi)$.*

Proof. Let ψ be any predicate with which q is quotient partitionable, and $\mathfrak{T}$ be any ψ-quotient partition for C (e.g., $\psi = \mathit{true}$ and $\mathfrak{T} = \{[\![C]\!]\}$). Define the non-relational property $P(\pi) = \pi \in [\![C]\!]$. Then, we have $\mathsf{RBPS}(P, q)$ and $\forall \pi \in [\![C]\!] \cap T.P(\pi)$ for every T. □

Completeness is arguably only of theoretical interest because, as shown in the proof, it holds independently of partition choices and can use somewhat "unfair" non-relational properties. (The proof essentially says that one can always verify q non-relationally by checking the trace containment against a program that is known to satisfy q).

Relational partition properties. While we have focused our efforts on properties for each partition that are non-relational, it is worth noting that this is not necessary: having relational properties instead would still work. For timing attacks, as explained earlier, we found that focusing on non-relational properties is a good way to avoid explosion of the state space.

More precisely, we can extend the notion of relational-by-property-sharing so that for Θ, an m-ary relation, $\mathsf{RBPS}(\Theta, q)$ is the condition below.

$$\forall \pi_1, ..., \pi_k. \\ \bigwedge_{\{\pi'_1, \ldots, \pi'_m\} \subseteq \{\pi_1, \ldots, \pi_k\}} \Theta(\pi'_1, \ldots, \pi'_m) \Rightarrow \Phi_q(\pi_1, \ldots, \pi_k)$$

Such relational properties Θ can replace the relational-by-property-sharing non-relational properties P in our framework. In other words, if q is ψ-partitionable, and $\mathfrak{T}$ is a ψ-quotient partition for C, then, $q(C)$ holds if for each $T \in \mathfrak{T}$, there exists Θ_T such that $\mathsf{RBPS}(\Theta_T, q)$ and for each $\pi_1, \ldots, \pi_m \in [\![C]\!] \cap T$, $\Theta_T(\pi_1, \ldots, \pi_m)$. Furthermore, such properties Θ_T can be analysed in a manner similar to Φ_q, if needed.

3.4 More General k-Safety Examples

The framework we presented above is general enough to apply to k-safety properties other than timing channel freedom. We conclude this section with some examples.

2-safety. First, there are other commonly used 2-safety properties on which our decomposition method can be applied. One is *determinism*, expressed as

$$\begin{aligned} \mathit{det}(C) \triangleq \forall\, \pi_1, \pi_2.\\ \quad in(\pi_1) = in(\pi_2) \Rightarrow out(\pi_1) = out(\pi_2). \end{aligned}$$

For $\psi_{\mathit{det}}(\pi_1, \pi_2)$ defined as $in(\pi_1) = in(\pi_2)$, the property $\mathit{det}(C)$ is ψ_{det}-quotient partitionable. Then, for any function g over the domain of the inputs of the traces, the property $P_g(\pi) \triangleq out(\pi) = g(in(\pi))$ is such that $\mathsf{RBPS}(P_g, \mathit{det})$ holds. It follows from Theorem 3.1, that a program C is deterministic, if we can find a partition $\mathfrak{T}$, where for each $T \in \mathfrak{T}$, there exists a function g_T such that $\mathsf{RBPS}(P_{g_T}, \mathit{det})$ and for each $\pi \in [\![C]\!] \cap T$, $P_{g_T}(\pi)$.

k-safety, k > 2. The examples we have focused on until now, are for the most part 2-safety properties, but this framework is developed generally for k-safety properties where k can be larger than 2. One such property is the *channel capacity* property [4, 25, 32, 41], which is a generalization of timing channel freedom. Note that the timing-channel freedom is defined to be the property that there can only be *one* running time (up to some small attacker-unobservable fluctuation) per public input. The channel capacity property is a relaxation which says that there are at most q (for some fixed $q \in \mathbb{Z}$) running times per public input. For instance, the property can be formally written as follows for $q = 2$:

$$\begin{aligned} \mathit{ccf}(C) \triangleq \forall\, \pi_1, \pi_2, \pi_3 \in [\![C]\!]^3.\\ \big(in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}] = in(\pi_3)[\boldsymbol{\ell}]\big) \Rightarrow\\ \big(time(\pi_1) \approx time(\pi_2) \vee\\ time(\pi_1) \approx time(\pi_3) \vee\\ time(\pi_2) \approx time(\pi_3)\big). \end{aligned}$$

The channel capacity property is a k-safety property for $k = q + 1$ (in fact, the bound is tight [42, 43]), and the timing channel property that we have been looking at closely is a specialization where $q = 1$. Notice that this k-safety property is ψ_{ccf}-quotient partitionable with $\psi_{\mathit{ccf}}(\pi_1, \pi_2, \pi_3) \triangleq in(\pi_1)[\boldsymbol{\ell}] = in(\pi_2)[\boldsymbol{\ell}] = in(\pi_3)[\boldsymbol{\ell}]$, and the analysis closely resembles the one for the property $\mathit{tcf}(C)$ described earlier. In particular, following the same line of thought as Example 7, properties $P_{f_1,f_2}(\pi)$ for high-independent functions f_1 and f_2, where $P_{f_1,f_2}(\pi)$ holds iff the running time of π is close enough to $f_1(\pi)$ or $f_2(\pi)$, are such that $\mathsf{RBPS}(P_{f_1,f_2}, \mathit{ccf})$ holds.

4. Symbolic Partitioning with Trails

The concept of ψ-quotient partitioning is semantic and naturally leads to the question of abstraction. This section introduces *trails*, a symbolic representation with which we can iteratively construct a trace partition for proving timing channel freedom, or finding a counterexample.

4.1 Definition

Given a program C and its Control Flow Graph (CFG) G_C, a trail expression tr is a regular expression where characters represent the edges of G_C, such that $L(tr)$ is the set of strings over the alphabet of the edges of G_C that satisfy tr. We call both the expressions as well as the set of strings they define as *trails*, and use subscripted symbols tr for the expressions and $L(tr)$ to denote the set they define. Syntactically, a trail tr is defined: $tr ::= \epsilon \;\|\; b_1b_2 \;\|\; tr_1|tr_2 \;\|\; tr_1 \cdot tr_2 \;\|\; tr^*$, where $L(\epsilon)$ is the trail containing just the empty string, and for each $L(b_1b_2)$ represents a single letter of the alphabet corresponding to the edge of G_C from the block b_1 to the block b_2. The definition of L is defined inductively in the usual way. For example $L(tr_1|tr_2)$ is the set of strings that are either in $L(tr_1)$ or $L(tr_2)$.

Given the CFG G_C of a program C, we define the *control flow graph automaton* A_C of C to be the automaton $(Q, \Sigma, \delta, q_0, F)$ over the alphabet $\Sigma = E(G_C)$ with set of states Q being the blocks of G_C and where a transition exists form a state q to a state p with letter (q, p) exactly when there is an edge e in G_C from the block q to the block p. The initial state q_0 is the entry block of G_C and the set of final states F is a singleton containing the exit block of G_C.

Given a program C, we say a trail is a *most general trail* of C, denoted tr_{mg}, if $L(tr_{mg}) = L(A_C)$, where $L(A_C)$ is the language recognized by the Control Flow Graph Automaton A_C of C. Notice that the set of traces described by a tr_{mg} of C is always a (not necessarily strict) superset of the actual possible traces of C.

4.2 Annotating Trails with low and high

We now describe how we annotate the constructors of a trail, in particular the union ('|') and Kleene star ('*') constructors, as low-dependent and/or high-dependent. As the procedure is the same for both low and high variables and dependent blocks, we consider only low-dependent blocks below.

We assume a taint-analysis module that, given a program C, returns a list of the variables in C and the blocks in the CFG G_C of C that are low-dependent, i.e., *tainted*. A block can only be tainted if it is branching in some way. As a result, there are exactly two outgoing edges from the block. We assume a similar module for high-dependent variables and blocks.

Suppose that a block b of the CFG G_C is marked as low -dependent, and let (b, b_1) and (b, b_2) be the two outgoing edges from block b. Then the constructor '$|$' in a trail $tr_1|tr_2$ is low-dependent with respect to b, if it is the outermost union constructor such that for at least one of the two tr_i's, one of the edges from b appears in the set of traces defined by it, whereas the other edge does not. Similarly, the construct '*' of a trail tr^* is marked as low-dependent with respect to b, if it is the outermost Kleene star constructor where one of the edges (b, b_i) is in the set of traces $L(tr)$ and the other edge is not. Formally, we have a recursive definition as to how a trail constructor is marked as low/high-dependent (omitted for lack of space).

If a union constructor of a trail is low-dependent, high-dependent or both, we write '$|_l$', '$|_h$' or '$|_{l,h}$' respectively. Similarly for '*' constructs we write '$^*_l$', '$^*_h$' or '$^*_{l,h}$' for the corresponding scenarios.

4.3 Partition Refinement

We now describe a technique that, given a ψ_{SC}-quotient partition $\mathfrak{T}$, constructs a new partition $\mathfrak{T}'$ that is also ψ_{SC}-quotient, using a collection of pluggable strategies. In what follows, we call a partition $\mathfrak{T}$ *safe* if it is a ψ_{SC}-quotient partition for $\psi_{SC}(\pi_1, \pi_2) \triangleq in(\pi_1) =_{low} in(\pi_2)$ and we call it *vulnerable* otherwise. A partition $\mathfrak{T}'$ is a refinement of a partition $\mathfrak{T}$ if for all $T' \in \mathfrak{T}'$ there is $T \in \mathfrak{T}$ such that $T' \subseteq T$.

One of the main goals of the algorithm is to produce ψ_{SC}-quotient partitions. We do this using the annotations on the trails and by representing a partition as a tree of trails $tr_1, \ldots, tr_n$, such that a trail tr_i is a child of a trail tr_j only if $L(tr_i) \subseteq L(tr_j)$. Any set of nodes of the tree with trails $tr_1, \ldots, tr_n$, forms a trail partition if $L(tr_1) \cup \ldots \cup L(tr_n) \supseteq L(tr_{mg})$. As we described in Section 3, there is no need for the partition components to be disjoint from each other, and we don't enforce such disjointness on the trails associated with the different nodes of the tree.

Suppose we have a trail $tr = tr_1 \mid_l tr_2$, where the union constructor has been annotated as low-dependent with respect to some low variables $l_1, \ldots, l_m$. Consider then the partition $\{L(tr_1), L(tr_2)\}$ of $L(tr)$. This partition can be seen to be ψ_{SC}-quotient for the following reason. Suppose we have two traces π_1 and π_2. According to the definition of ψ_{SC}-quotient partition, we want to make sure that if the low inputs of these two traces are the same, then they are both in $L(tr_1)$ or both in $L(tr_2)$. If this is the case though, it means that both traces agree on the variables $l_1, \ldots, l_m$, and thus all traces will follow the same CFG edge, which in turn implies that both π_1 and π_2 will either be in $L(tr_1)$ or in $L(tr_2)$.

We can apply a similar strategy for Kleene star constructors. We also take a similar approach for trails dependent on high data. We do this not when creating a partition to prove side-channel absence, but rather when trying to synthesize a possible attack when such a proof has failed.

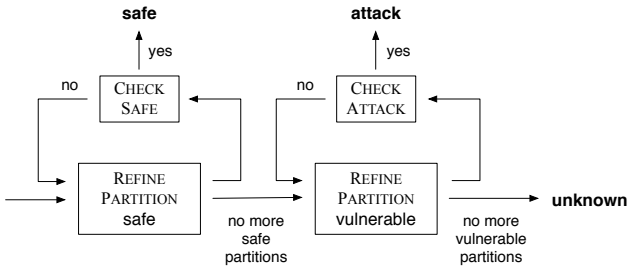


Figure 2. The overall algorithm.

4.4 Algorithm

The main procedure receives a program C as input, as well as information about which variables and CFG blocks are low/high (from a taint analysis) and attempts to prove safety. Failing at that, the algorithm tries to produce an attack specification. We further assume two procedures: ANNOTATETRAIL(tr) uses the annotated blocks of G to annotate the trail constructors of the input trail tr (Section 4.2); BOUNDANALYSIS(tr) calculates symbolic lower and upper bound expressions for the traces described by the input trail tr (Section 5). The main subprocedures of our algorithm are:

- REFINEPARTITION($\mathfrak{T}$, {safe, vulnerable}): Produces a refined partition $\mathfrak{T}'$ of $\mathfrak{T}$ that is either safe or vulnerable, depending on the second argument. In Section 4.3, details are given on how the partitions are extracted by the tree of trails that is constructed and how the annotated trails obtained from ANNOTATETRAIL are used to ensure ψ-quotientability.
- CHECKSAFE($\mathfrak{T}_S$): Returns yes if the given partition is verifiably safe, and no otherwise. For each partition component, this procedure employs BOUNDANALYSIS, attempting to find symbolic upper and lower bounds for the traces in the component. It returns yes if there is no correlation between high variables and running time.
- CHECKATTACK($\mathfrak{T}_V$): Similarly, this procedure returns yes if there is a suspicion for an attack, and no otherwise. To this end, it attempts to either find a partition component where BOUNDANALYSIS calculates symbolic bounds that *are* correlated with high variables, or alternatively tries to find two components T_1 and T_2, such that $\mathfrak{T} = T_1 \uplus T_2$ is not a ψ_{SC}-quotient partition, and the symbolic bounds of the two components differ from each other (even if for each component individually the symbolic bounds obtained do not depend on high variables).

The overall flow is as follows (depicted in Fig. 2 and illustrated in Section 2). The initial input to the algorithm is the partition comprising only the component $[\![C]\!]$, represented by the most general trail tr_{mg}. The procedure goes into a first refinement loop, that (i) refines the partition into a safe one (on the first iteration the partition $\{[\![C]\!]\}$ is left unchanged), and (ii) sends the refined partition to the pro-

cedure CHECKSAFE. If the latter returns yes, the iteration stops and outputs that the program is verifiably safe. If not, then the loop continues until either CHECKSAFE returns yes (and thus outputs that the program is safe and the algorithm stops) or REFINEPARTITION cannot further refine the given partition into a safe one.

In the latter case, the algorithm enters a second refinement loop, aimed at discovering an attack. It starts by having REFINEPARTITION refine the given partition into a vulnerable one, and then passes the new refined partition to CHECKATTACK. Similarly, this loop continues until either CHECKATTACK returns yes or REFINEPARTITION fails to further refine the partition. In the first case, the algorithm outputs a set of possible attack specifications, and in the second case the algorithm simply indicates that it failed to produce a meaningful summary.

In a sense, given enough strategies for REFINEPARTITION the procedure can continue indefinitely, but in practice we provide the procedure with parameters around the size and form of the partitions produced, that would cause the program to exit earlier, and without giving a concrete answer as to whether the program is safe or whether there is a suspicion for an attack.

5. Implementation

We have implemented our technique as the *Blazer* tool for proving the absence of timing-channel vulnerabilities in Java bytecode programs. To process bytecode, *Blazer* uses the WALA front-end [39], which transforms bytecode into an SSA-based CFG. We used the information flow (taint) analysis JOANA [33] in order to annotate blocks as to whether branching depends on low (taint) or high (secret) variables.

For working with trails, *Blazer* uses the `brics` automaton library [13] to check language inclusion and construct intersection, union, and complementation automata. We have wrapped this library with a translation between trails (described as regular expressions) and automata.

We have built a custom abstract interpreter on top of WALA, using the Parma Polyhedra Library (PPL) [6] to compute numerical invariants. The abstract interpreter can be directed to restrict analysis to a given trail. We leverage the seeding technique [10] to compute transition invariants [30], and match these invariants against a database of complexity-bound lemmas [16, 17]. This embodied in the BOUNDANALYSIS component. While the abstract interpreter is interprocedural, BOUNDANALYSIS currently relies on manually-specified bound summaries for interprocedural function calls. In the future, these summaries will be computed automatically.

When comparing running times of two nodes in the trails tree, we need to know whether there is an observable difference between them. *Blazer* employs multiple approaches. We have a generic component that computes the highest degree of the complexity bound polynomial, a rough heuristic that works for many examples. In other cases, a platform-specific model of execution cost can be used. Here we make assumptions about the maximum values of the input variables to compute the concrete number of instructions a bound expression represents. Then the observable difference between bounds can be defined as a threshold distance in numbers of instructions. We currently use a simple machine model in which each bytecode instruction is counted as a single unit.

6. Evaluation

We evaluated the implementation of *Blazer* on 24 benchmarks, including 6 examples drawn from the DARPA Space/-Time Analysis for Cybersecurity (STAC) challenge problems [35] and 6 real-world programs in which timing attacks were exploited and reported in cryptography papers [15, 22, 29]. Benchmarks are paired up so that there are two versions: the "unsafe" version is expected to be vulnerable to timing-channel attacks while the "safe" one is not. For third-party benchmarks, we created safe versions by hand (except for User). Our experiment harness executes *Blazer* on both the safe and unsafe versions of each benchmark.

6.1 Benchmarks

Our benchmarks, which reflect a broad range of code patterns, are up to 100 basic blocks in size (details reported later in this section). Fig. 3 illustrates some selected examples.

- *MicroBench.* These are hand-crafted to exercise the various aspects of *Blazer*. We start with simple examples; `nosecret_safe` tests the basics of side-channel detection, which can only occur when there is a secret. The others are more intricate. The `loopAndBranch` benchmark, for instance, is shown in Fig. 3. At first this seems to have a vulnerability, but the potentially vulnerable trail is infeasible, which is caught by the abstract interpreter. Also shown in Fig. 3 is the classic Unix login vulnerability that leaks usernames. When line 7 is removed, the program takes longer when a username exists because it hashes the input password via `md5`.
- *STAC.* Several benchmarks were extracted from the DARPA Space/Time Analysis for Cybersecurity (STAC) engagement problems [35]. `modPow1` (shown in Fig. 3) and `modPow2` perform cryptographic arithmetic using the Java `BigInteger` library.
- *Literature.* We have also crafted examples taken from papers that demonstrate timing attacks on real-world cryptographic methods. These include Genkin *et al.* [15] (*GPT*), Kocher [22] (*K96*), and Pasareanu *et al.* [29] (*PPM16*). The example from *PPM16* is discussed in Section 2 and shown in Fig. 1.

Blazer supports manually-specified summaries of running times so we specify running times for library calls such as those to the Java `BigInteger` library (in `modPow#` and `Cryptography` benchmarks).

```
1 void loopAndbranch_safe(int high, int low) {
2   int i = high;
3   if(low < 0) { while(i > 0) i−−; }
4   else {
5     low = low + 10; // low is above 0
6     if(low >= 10) { int j = high; while(j>0) j−−; }
7     else { if(high<0) { int k = high; while (k>0) k−−; } }
8   } }
```

```
1 boolean login_safe(String u, String p) {
2   boolean outcome = false;
3   if (map.containsKey(u)) {
4     if (map.get(u).equals(md5(p)))
5       outcome = true;
6   } else {
7     if (map.get(u).equals(md5(p))) { } // remove for unsafe
8   }
9   return outcome; }
```

```
1 BigInteger modPow1_safe(BigInteger base, BigInteger
      exponent, BigInteger modulus) {
2   BigInteger s = BigInteger.valueOf(1);
3   int width = exponent.bitLength();
4   for (int i = 0; i < width; i++) {
5     s = s.multiply(s).mod(modulus);
6     if (exponent.testBit(width − i − 1))
7       s = s.multiply(base).mod(modulus);
8     else s.multiply(base).mod(modulus); // remove for
          unsafe
9   }
10  return s; }
```

Figure 3. Some examples selected from the benchmarks. For lack of space, only the main methods are shown.

Observer modeling. As discussed in the implementation section, observable running time differences are modeled in several ways. We designed the *MicroBench* so that distinguishing between safe and unsafe is possible by evaluating computational complexity, *e.g.*, linear vs. quadratic. The variables are assumed to be unbounded, while a safe program is assumed to be one where the symbolic running times have the same polynomial degree. While sufficient for these hand-crafted micro-benchmarks, this model of observability is too simplistic for real-world code.

For the real-world examples from *STAC* and *Literature*, we use a model of observable running time based on concrete differences in bytecode instructions between partitions. We assume some reasonable maximum for the input variables, *e.g.*, 4096 bits for the cryptographic benchmarks. Then we plug these values into the symbolic bound expressions to get a concrete estimate of the maximum number of bytecode instructions. Using this method, an observable difference is defined by some minimum threshold in the difference between the number of instructions. For these benchmarks, we use a low number of instructions (25k) to define the observable difference in running time. In real-world applications of this verification, observability depends on many factors, including hardware, operating system, network latency, etc, and would need to have application-specific calibration.

Benchmark	Size	Safety Time (s)	w/Attack Time (s)
MicroBench			
`array_safe`	16	1.60	–
`array_unsafe`	14	0.16	0.70
`loopBranch_safe`	15	0.23	–
`loopBranch_unsafe`	15	0.65	1.54
`nosecret_safe`	7	0.35	–
`notaint_unsafe`	9	0.28	1.77
`sanity_safe`	10	0.63	–
`sanity_unsafe`	9	0.30	0.58
`straightline_safe`	7	0.21	–
`straightline_unsafe`	7	22.20	28.49
`unixlogin_safe`	16	0.86	–
`unixlogin_unsafe`	11	0.77	1.27
STAC			
`modPow1_safe`	18	1.47	–
`modPow1_unsafe`	58	218.54	464.52
`modPow2_safe`	20	1.62	–
`modPow2_unsafe`	106	7813.68	31758.92
`pwdEqual_safe`	16	2.70	–
`pwdEqual_unsafe`	15	1.30	2.90
Literature			
`gpt14_safe`	15	1.43	–
`gpt14_unsafe`	26	219.30	1554.64
`k96_safe`	17	0.70	–
`k96_unsafe`	15	1.29	3.14
`login_safe`	18	6.54	–
`login_unsafe`	17	4.40	9.10

Table 1. The results of applying our tool *Blazer* to a variety of benchmarks: hand-made examples, examples from the literature [15, 22, 29], and extracted from DARPA STAC challenge problems. Median running times are shown for safety verification alone as well as safety verification plus the search for an attack specification. The safe benchmarks need no search for attack specification, so they have – instead of a running time.

6.2 Results

The benchmarks were run sequentially on a single commodity PC with a quad-core 3.07GHz processor and 12GB of RAM. Running time is collected for both safety verification alone as well as safety verification plus the search for an attack specification. The latter running time only applies to the unsafe benchmark, since the tool halts if it proves safety. Running time is measured by performing five runs and taking the median.

Table 1 shows the results. The **Benchmark** column identifies the benchmark's method and alternates between the safe and unsafe versions. **Size** indicates the number of basic blocks in the method's control-flow graph. The **Safety Time** column shows the tool's median running time in seconds for safety verification alone, while the **w/Attack Time** column is the median running for safety verification and the subsequent search for an attack specification.

Our tool is sound: it either determines the program is safe, finds an attack specification, or gives up. For every safe benchmark, *Blazer* verified the safety of the benchmark. In all unsafe benchmarks, the tool found an attack specification, i.e., two candidate subtrails with differing running times, except for `gpt14_unsafe`.

For most benchmarks, the safety verification takes only a few seconds, save some notable outliers. The running time for generating an attack specification, which includes the safety verification, often takes longer, because it is a computationally more intensive step. It takes the trails tree output from safety verification and further decomposing it into subtrails.

As for the outliers, the running time appears loosely related to the number of basic blocks in the program, as shown by the very high running times of the outliers `modPow1_unsafe`, `modPow2_unsafe`, and `gpt14_unsafe`. This is due to a combinatorial explosion of subtrails, superlinear with respect to the number of conditional branches, as well as the increased memory pressure of storing and processing the tree o decomposed subtrails. The running time for `straightline_unsafe` is an exception to this relationship: it has few basic blocks but a long running time. This is likely due to a particularly large basic block that has 90 instructions, increasing the processing time of the subtrails that contain it.

7. Related work

To the best of our knowledge, our approach is the first method that tackles the timing channel freedom (TCF) verification problem by iteratively decomposing the problem into sub-problems that require non-relational reasoning.

Type-based enforcement. TCF is a kind of *noninterference*: no matter the secret inputs to the program, the running times observable by adversary when given the same set of non-secret inputs are (roughly) the same. Noninterference properties can be enforced by type systems. For example, work by Agat [1] uses a Volpano-Irvine-Smith-style non-interference type system [38] to transform a given program into one without timing channels (basically, by inserting *padding* no-op instructions in places identified by the type system). The type-based approach can also be used to check timing channel security, as shown by Hedin and Sands [19]. Type-based enforcement is efficient but often imprecise, falsely sounding the alarm on correct programs. For example, here are two trivial examples that cannot be typed with such approaches even though they are secure:

```
1    void ex1(x,h) { if false { while (h < x) { h++; } }}
2    void ex2(x,y) {
3        if (h > x) { tick; } else { tick;tick;}
4        if (h <= x) { tick; } else { tick;tick; }    }
```

Our technique is able to prove that these examples safe because, through our use of trails, we can determine feasibility of different path cases. A more elaborate example is the `loopAndBranch` micro-benchmark, shown in Fig. 3 and discussed in Section 6. The safety of programs such as `loopAndBranch` can depend on subtle conditions and our strategy of decomposition enables us to truly leverage invariants from abstract interpretation to this end.

Quantitative analysis. *Quantitative information flow* (QIF) is a quantitative generalization of non-interference founded on rigorous information-theoretic principles [4, 25, 32, 41] (a channel is non-interferent iff its quantitative information flow is zero). Recently, researchers have sought to apply QIF to side channel security [14, 23, 29, 45]. For instance, Zhang *et al.* [45] propose an approach that ensures the boundedness of the quantity of information leaked by the timing channel. Their approach addresses low-level issues such as cache uses, and comprises type-based analysis, hardware assistance, and predictive timing mitigation. The quantitative approach is also taken in the work by Doychev *et al.* [14] on a static analysis for detecting leaks due to cache uses, and the work by Pasareanu *et al.* [29] that quantifies information leak amount via symbolic execution and model counting. Due to its under-approximate nature, Pasareanu *et al.*'s technique is more suited to finding possible violations of TCF, rather than verifying it.

Our analysis does not address the quantitative aspect of timing channel security. However, ideas from our approach may apply to them [42, 43], when viewed as k-safety problems (discussed more below). Moreover, our approach may complement Pasareanu *et al.*: Our tool emits attack specifications that could be used to construct a slice of the program where there is likely to be an attack, which can given as input to their tool.

Self-composition. Generally speaking, TCF is a 2-*safety property* in that it relates 2 executions. In particular, it relates all pairs of executions such that if they agree on non-secret inputs then they (roughy) agree on running time. A general approach to verifying k-safety properties (which relate $k \geq 2$ executions) is to use *self-composition* [7, 12, 27, 36, 37, 43]. The approach reduces the k-safety problem to a 1-safety problem by composing (sequentially, in parallel, or in an interleaving fashion [27, 36, 37]) k copies of the given program prior to applying a (standard) verification technique. More precisely, the k-safety problem $\forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k.\Phi(\pi_1, \ldots, \pi_k)$ is reduced to the 1-safety problem $\forall \pi \in [\![C^{(k)}]\!].\Phi(\pi_{(1)}, \ldots, \pi_{(k)})$ where $C^{(k)}$

is the composed program whose execution trace is a k-tuple of (copies of) C's execution traces and $\pi_{(i)}$ projects the i-th trace in the composite trace π. The approach is clearly sound and complete for arbitrary k-safety properties, relative to the soundness and completeness of the backend (1-)safety verifier. However, it trades robustness for scalability due to the rather brute force reduction, and the naïve self-composition approach only scales to relatively simple examples when paired with an automatic backend safety verifier.

Almedia *et al.* [2] have pursued the self-composition-based approach to timing channel security verification in particular, with mixed results. More recently, they have proposed to check if a program is *constant-time* by self-composition [3]. A program being constant-time is a stronger requirement than timing channel security, and it requires the program's control flow to be independent of the high security data. The strict requirement can be exploited to form a *tightly coupled self-composition*, thereby allowing efficient verification [27, 36, 37] (further discussed below).

Other methods for relational reasoning. Other methods have also been considered for verifying k-safety (and, more generally, relational) problems. Closely related to the self-composition approach is the recent work by Sousa and Dillig [34]. Similar to self-composition, their approach (implicitly) creates k copies of the program, but attempts to synchronize the verification's reasoning process so that it keeps the program flow of the different copies in lockstep as much as possible. This is done via intricate program logic rules that they call *Cartesian Hoare Logic*. Similar to the interleaving self-composition technique [27, 36, 37], such a lockstep reasoning improves the performance of the approach by more tightly coupling the key invariants across the program copies. Relational program logics pre-date Cartesian Hoare Logic and can be traced back to the product programs of Reynolds [31]. More recently, Benton introduced relational logics for program transformations and equivalence [9], and Yang provided a relational version of separation logic [40]. Other recent uses of product programs for program equivalence include [28, 44]. Also, a recent work by Assaf *et al.* [5] proposes to verify k-safety (and more general hyper-safety) problems via an abstract interpretation over program trace sets (as opposed to traces), which may be understood as an abstract interpretation over a (possibly unbounded) product of program copies. Recent work by Çiçek *et al.* on *relational cost analysis* [11] describes a program analysis for checking and inferring resource-usage properties spanning multiple programs. Such an analysis can potentially be used for proving timing channel security by applying it to the program copies.

In contrast to self composition and its related variants discussed above, the method proposed in this paper does not (explicitly or implicitly) make program copies and reduce the problem to a 1-safety problem. Instead, it decomposes the given program by utilizing a certain decomposition property of the target k-safety problem that we call ψ-*quotient partitionability*. This way, we obtain the answer for the whole by solving the decomposed sub-problems. Note that, in our approach, a non-relational analysis (*i.e.*, 1-safety verification) is not applied to the self-composed program that completely expresses the original k-safety problem, but instead, it is applied to decomposed sub-problems of the original.

Path sensitivity. Our trails act as specifications for how a program can be restricted based on certain branching choices. This is similar to path sensitivity approaches such as trace partitioning [26]. Similarly, PAGAI [21] employs a technique called path focusing [20] to guide an abstract interpreter to consider certain paths at a time. In comparison to control-flow refinement (CFR) [17] our trails are a generalization: we allow more complicated forms of loop unrolling (arbitrary regexes) and trails may *eliminate* executions whereas CFR always encompasses all executions. To our knowledge, none of the above approaches have been used to reason about relational properties, permit low-vs-high annotations, nor define a decomposition that reduces a relational property to trail/path (*i.e.*, trace) properties.

8. Conclusion

We have presented a novel technique for proving that a program is free of timing channels. Our approach, which we have proved sound, is based on a general decomposition technique that reduces k-safety verification (timing channel freedom has $k = 2$) to a task fit for a non-relational analysis. Our decomposition proceeds iteratively by symbolically synthesizing path-based partitions ("trails") that case-split in instances of taint-based or secret-based branching. The former is used to prove timing channel freedom, while the latter helps identify particular vulnerabilities. We implemented our approach in a tool called `Blazer` and demonstrated its effectiveness on a collection of 25 benchmark examples, including 6 drawn from the STAC challenge problems and 7 from the cryptography literature. We believe that our decomposition sets the stage for the development of more powerful relational analyses and/or other k-safety property verifiers.

Acknowledgments. We thank Sharon Shoham, our shepherd, for helpful comments and suggestions. This research was supported in part by the National Science Foundation under grant CNS-1314857; by MEXT Kakenhi 17H01720, and JSPS Core-to-Core Program, A.Advanced Research Networks; and by DARPA under contracts FA8750-15-2-0104 and FA8750-16-C-0022.

References

[1] J. Agat. Transforming out timing leaks. In *POPL*, 2000.

[2] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7), 2013.

[3] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, 2016.

[4] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In *CSF*, 2012.

[5] M. Assaf, D. A. Naumann, J. Signoles, E. Totel, and F. Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *POPL*, 2017.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Compututer Programming*, 72(1-2), 2008.

[7] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[8] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[10] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[11] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. In *POPL*, 2017.

[12] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *SPC*, 2005.

[13] dk.brics.automaton. Finite-state automata and regular expressions for Java. `http://www.brics.dk/automaton/`, 2017.

[14] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, 18(1), 2015.

[15] D. Genkin, I. Pipman, and E. Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. In *CHES*, 2014.

[16] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, 2010.

[17] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.

[18] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[19] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2005.

[20] J. Henry. Static analysis by path focusing. Master's thesis, Grenoble INP, 2011.

[21] J. Henry, D. Monniaux, and M. Moy. PAGAI: A path sensitive static analyser. In *TAPAS*, 2012.

[22] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[23] B. Köpf and D. A. Basin. Automatically deriving information-theoretic bounds for adaptive side-channel attacks. *Journal of Computer Security*, 19(1), 2011.

[24] S. Langkemper. The password guessing bug in Tenex. `https://www.sjoerdlangkemper.nl/2016/11/01/tenex-password-bug/`, 2016.

[25] P. Malacaria. Assessing security threats of looping constructs. In *POPL*, 2007.

[26] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.

[27] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS*, 2006.

[28] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *SAS*, 2013.

[29] C. S. Pasareanu, Q. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-SMT. In *CSF*, 2016.

[30] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

[31] J. C. Reynolds. *The Craft of Programming*. Prentice Hall International series in computer science. Prentice Hall, 1981.

[32] G. Smith. On the foundations of quantitative information flow. In *FOSSACS*, 2009.

[33] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab. Checking probabilistic noninterference using JOANA. *it - Information Technology*, 56(6), 2014.

[34] M. Sousa and I. Dillig. Cartesian Hoare logic for verifying k-safety properties. In *PLDI*, 2016.

[35] STAC. DARPA space/time analysis for cybersecurity (STAC) program. `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`, 2017.

[36] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[37] H. Unno, N. Kobayashi, and A. Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference. In *PLAS*, 2006.

[38] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4 (2/3), 1996.

[39] WALA. IBM T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/`, 2017.

[40] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3), 2007.

[41] H. Yasuoka and T. Terauchi. Quantitative information flow - verification hardness and possibilities. In *CSF*, 2010.

[42] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. *Journal of Computer Security*, 19 (6), 2011.

[43] H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theoretical Computer Science*, 538, 2014.

[44] A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *FM*, 2008.

[45] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.