Decomposition Instead of Self-Composition for k-Safety

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

Yale University [†] University of Maryland [‡] JAIST

Abstract

We describe a novel technique for proving k-safety properties (non-interference, determinism, etc.) via a decomposition that enables one to leverage non-relational reasoning techniques. The key is the inter-operation of the following principles. First, we observe that many k-safety properties of interest have a particular structure that we call *ψ-quotient partitionability* where ψ is a k-ary formula. Second, we develop a partitioning strategy of execution traces based on the k-safety property Φ of interest such that if ψ holds for k traces then they must be in the same partition. Finally, within a partition component T_i, we observe that we can prove k-safety by instead proving a universal property: all traces within the partition satisfy some common property P_i, chosen to be strong enough that it implies the k-safety property Φ of any k-tuple of traces in components T_i.

We apply this strategy to the task of discovering timing side channels. A key feature of our approach is a demand-driven partitioning strategy that uses high/low-annotated regex-like *trails* to reason about one partition component of execution traces at a time. We have applied our technique in a prototype implementation tool called *Blazer*, based on WALA, PPL, Z3, and the brics automaton library. We have proved non-interference of (or synthesized an attack specification for) 25 programs written in Java bytecode, including 7 classic examples from the literature, and 6 examples extracted from the DARPA STAC challenge problems.

1. Introduction

Validating a *k-safety property* is a difficult problem. Such a property involves k runs of a program and, consequently, validation requires establishing relationships between k different execution traces. One example property we are interested in is the *absence of timing channels*. This is a 2-safety property: we want to show that no variation in timing can be observed in any pair of traces whose computations use the same public inputs but differing secrets, *e.g.*, passwords or encryption keys. In short, we want secrets' values and timing to be uncorrelated.

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 [3, 23]: 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, including clever improvements on the basic idea [4], can be expensive due to state space explosion. More recently, Sousa and Dillig [22] proposed *Cartesian Hoare Logic* (CHL) which allows stating and proving k-wise relational properties. The structure of the logic permits more efficient validation. Nonetheless, even CHL relies, at least implicitly, on making k copies of the program, which means that invariants are split across the product program and key information is lost during fixpoint computation.

This paper. We depart from the composition-based strategies and instead establish a novel *decomposition* methodology that has the potential to check some k-safety properties more efficiently than past work. It involves an idea we call *ψ-quotient partitionability*, which we now explain.

Formally, a *k-safety property* [6, 23] can be stated as follows. Let C be a program, and $[\![C]\!]$ be the set of execution traces of C. Then, a k-safety property has the form $\forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k.\ \Phi(\pi_1, \ldots, \pi_k)$. For timing channels (a 2-safety property), Φ_{SC} would have the form: $in(\pi_1) =_{low} in(\pi_2) \Rightarrow time(\pi_1) \approx time(\pi_2)$. That is, if the public (*i.e.*, low security) input variables of the two traces are the same (while the secrets can vary arbitrarily), then the observed running time is the same (up to the limits of the observer).

Proving a property Φ using our approach has three steps:

1. Define a *quotient formula* ψ over k traces $\pi_1, ..., \pi_k$. For side-channel freedom, we let quotient $\psi_{SC}(\pi_1, \pi_2) \triangleq in(\pi_1) =_{low} in(\pi_2)$.
2. Define a method of decomposing sets of execution traces of program C into a *ψ-quotient partition* $\mathfrak{T}$. Formally, the

method should produce a partition $\mathfrak{T} = T_1, ..., T_n$ (for some $n \geq 1$) of the set of all execution traces of C such that for any k traces $\pi_1, ..., \pi_k$, if $\psi(\pi_1, ..., \pi_k)$ holds then there exists some $T_i \in \mathfrak{T}$ such that $\{\pi_1, ..., \pi_k\} \subseteq T_i$.[1] For timing channels, we use *tainting* (the influence of public inputs) to partition the traces; doing so ensures that traces involving equal public inputs are in the same partition.

3. Define a set of properties P_i, one for each bucket T_i, such that if $P_i(\pi)$ holds for all $\pi \in T_i$ then $\Phi(\pi_1, ..., \pi_k)$ holds for all $\{\pi_1, ..., \pi_k\} \subseteq T_i$. Properties P_i are selected from a class of properties determined by the k-safety property Φ of interest, irrespective of the program C. In the case of side channel absence, the P_i relate to running time: $P_i(\pi) \triangleq \mathit{time}(\pi) = f(\mathit{in}(\pi)[\mathsf{low}]) \pm c$; *i.e.*, the running time of each trace is a function f of the public (*i.e.* low security) input, plus or minus some constant c up to the limits of the observer.

A key result of this paper is that the outcome of these three steps—the quotient ψ, the partition $\mathfrak{T} = T_1, ..., T_n$, and per-partition properties $P_1, ..., P_n$—is sufficient to prove k-safety property Φ. For timing channels, this should be evident: Each trace in a partition component T_i has the same running time function f over low security variables (due to P_i); traces in different partitions have unequal public inputs (due to Φ_{SC}) and thus satisfy Φ_{SC} vacuously. In short, *k-quotient partitionability* is sufficient to prove that *all* k-tuples satisfy Φ by proving that each trace within some T_i satisfies P_i (*e.g.*, by using abstract interpretation). We do not have to reason across multiple traces, whether within or between partition component, thus avoiding the state space explosion of the self-composition approach.

Synthesizing partitions for timing channels. A key element of our approach is synthesizing the partition (and the corresponding per-component properties) iteratively. We do this using an abstraction called *trails*, which describe sets of executions. A trail t_i is an annotated regular expression (regex) over the edges of the control-flow graph, and specifies a restriction to a subset of the traces of the program.

For proving the absence of timing channels, our partitioning strategy is based on the influence of 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 (as a function f over low security variables) described by the trail [9–11] by matching transition relations with a database of lemmas [10]. 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. Then we repeat the process. This same approach works for more general k-safety properties, too, as we discuss in Section 3.

New tool for k-safety. We have implemented our algorithm as a new tool called *Blazer*. To implement the algorithm described above, we equip an off-the-shelf abstract interpreter with the ability to consult an oracle (the synthesized trails) to decide which CFG arcs to follow, thus achieving partition-specific invariants. These invariants underlie the per-trail running time analysis.

When *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. We evaluated our tool on a collection of 25 benchmarks, including 12 tricky hand-crafted benchmarks, 7 programs from the literature [8, 15, 19], and 6 fragments of the DARPA STAC challenge problems[2] (Sec. 7). We show 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.

Contributions. In summary, this paper contributes:

1. A novel program analysis technique for k-safety properties that decomposes the problem into non-relational subproblems rather than resorting to relational abstract domains (Sec. 3).
2. A proof of soundness (Sec. 3).
3. A novel symbolic representation of partitions and a method for constructing new partitions (Sec. 4).
4. An algorithm for proving safety or discovering attack specifications (Sec. 5) and an implementation, embodied in the tool *Blazer* (Sec. 6).
5. An evaluation on 23 benchmarks, including examples from the STAC challenge problems and the cryptography literature [8, 15, 19] (Sec. 7).

Limitations. We require that the k-safety property Φ of interest be, what we call, ψ-quotient partitionable. For every Φ there is a trivial ψ-quotient: true. The question is whether there is a ψ that leads to a meaningful and useful quotient. In this paper we demonstrate that there is a useful ψ for non-interference and, hence, timing side-channels. It remains to be seen whether there are meaningful ψ quotients for other k-safety properties.

[1] Technically speaking, 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 not important.

[2] http://www.darpa.mil/program/space-time-analysis-for-cybersecurity

Our tool currently does not support recursive functions. Our general theoretical results apply nonetheless to such programs. To add support for recursion, one avenue forward would be to apply our annotation strategy—that is: marking nondeterministic constructors as to whether they represent high or low inputs—to the domain of context-free grammars. We leave this to future work.

2. Overview

In this section, we give a tour of our work with some examples. Our formal results (Section 3) apply to k-safety properties in general. To build intuition and motivation, we will focus on the problem of proving the *absence of timing channels* (an instance of 2-safety) in this section. We begin with simple examples to explain the key idea and then later show how it applies to more realistic examples.

For clarity, we begin with some (informal) definitions. Suppose an attacker has full knowledge of a program C running at some remote location, and can provide input low (taint) to the program C. At the same time, the program's execution is parameterized by some secret value high. The program is said to have a timing channel if the attacker can infer information about the secret value high by observing the running time of C for each value low. Programs may have other *side channels* beyond timing; *e.g.*, secrets may be leaked by observable variations in memory usage, power usage, etc.

The absence of timing channels (and, more generally, side channels) in a program is a k-safety property and more specifically a 2-safety property: for all traces π_1 and π_2, that are both executions of the program with the same (possibly attacker-controlled) input low, the externally observable running time of π_1 is roughly the same as that of π_2. Some differences in timing are permitted due to limitations on the observational power of the adversary.

Decomposition: Exploiting symmetries. To develop intuition for our technique, we consider the following example program:

```
void foo(int high, uint low) {
  if (high == 0) { i = 0; while(i < low) i++; }
  else { i = low; while(i > 0) i−−; }
}
```

The execution of this program depends 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 [3, 23] or a relational abstract domain [22]) 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 low. In this case we can be more specific: the running time is *linear* in low. We might write this as a property $P^{\text{lin}}_{\mathsf{low}}$, and then write:

$$\forall \pi \in \mathcal{T}.\ P^{\text{lin}}_{\mathsf{low}}(\pi)$$

where $\mathcal{T}$ is the set of all execution traces of the program. 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 low-based 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 $\mathcal{T} = \bigcup_{i\in[1,n]} T_i$) 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 the above example, we only needed one partition component. Here is another example:

```
void bar(int high, int low) {
  int i;
  if (low > 0) { // O(2∗low)
    i = 0; while(i<low) i++;
    while(i>0) i−−;
  } else { // O(1)
    if (high == 0) { i = 5; } else { i = 0; i++; }
  }
}
```

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 $\mathcal{T} \subseteq T_> \cup T_\leq$ 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. (Later we will describe how we symbolically synthesize this and more elaborate, control-flow-level partitions.) With some help (discussed below), we can coërce an off-the-shelf abstract interpreter to now perform two analyses proving, respectively, that:

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

Here, property $P^{\text{const}}_{\mathsf{c}}$ means that the running time is constant in low (for some fixed constant d), within some bound c that is not observable to the attacker. For this example, the constant bound $c = 2$; *i.e.*, the running time is basically 0, 1, or 2 instructions, and we assume such small differences are not significant to the attacker.

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 traces from two different partition components have different observations, this observation cannot be correlated with high security data.

A more realistic example. The program pwcheck1 in Fig. 1 is based on real-world vulnerable code [16]. Also included is a variant pwcheck2 that attempts to resolve the vulnerability and a final variant pwcheck3 that is safe. Program pwcheck1 has a side-channel vulnerability because the attacker can find a correlation between running time and the prefix of a correct guess of the password. We will now describe how our technique operates on these examples and is able to symbolically synthesize *attack specifications* (two static path descriptions with proofs that they have different runtime complexities) for pwcheck1 and pwcheck2, demonstrating the vulnerability, as well as a proof that pwcheck3 is not vulnerable.

Trails. Our trace partition in the previous example considered the difference input values (those for which low > 0, or not). Our full algorithm uses a different approach: We specify a partition by characterizing the branching taken by traces in each partition component. Paths are specified using annotated regular expressions tr that we call *trails*, which for the purposes of this example have the following grammar:

$$tr ::= \; ij \;\parallel\; i\!\downarrow \;\parallel\; tr_1 \cdot tr_2 \;\parallel\; tr_1|_\alpha\, tr_2 \;\parallel\; tr^*_\alpha$$

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, vertical bar for branching, and star for looping. Both branching and looping regex operators are annotated with l to mean low, h to mean high, or l, h to mean both. Here is an example:

$$23 \cdot 3\!\downarrow |_{l,h}\, (24 \cdots)$$

This represents a prefix of the execution of pwcheck1, focusing on the first **if**. To illustrate trails on source code, we use line numbers instead of CFG block numbers. Execution can flow from line 2 to line 3 and then exit directly, or it can flow from 2 to 4. The annotation on the branching operator indicates the **if** depends on low *and* high data (guess and pw).

Our technique begins by syntactically computing the *most general trail* tr_{mg} that covers all traces of the program; in this example:

$$tr_{mg} \triangleq 23 \cdot 3\!\downarrow |_{l,h}\, (24 \cdot 45 \cdot (57 \cdot 75)^*_{l,h} \cdot (57 \cdot 7\!\downarrow |_{l,h}\, 59 \cdot 9\!\downarrow))$$

The Kleene star expression identifies the **for** loop, whose iterations depend on low data (directly) and high data (indirectly). The final part of tr_{mg} says that execution can end either via the **return** statement within the loop on Line 7 or via the **return** statement on Line 9.

Overall algorithm. This most general trail tr_{mg} seeds our search for a partitioning. It describes all executions and our algorithm will iteratively search for a partition, described by a collection of trails $tr_1, \ldots, tr_n$ such that:

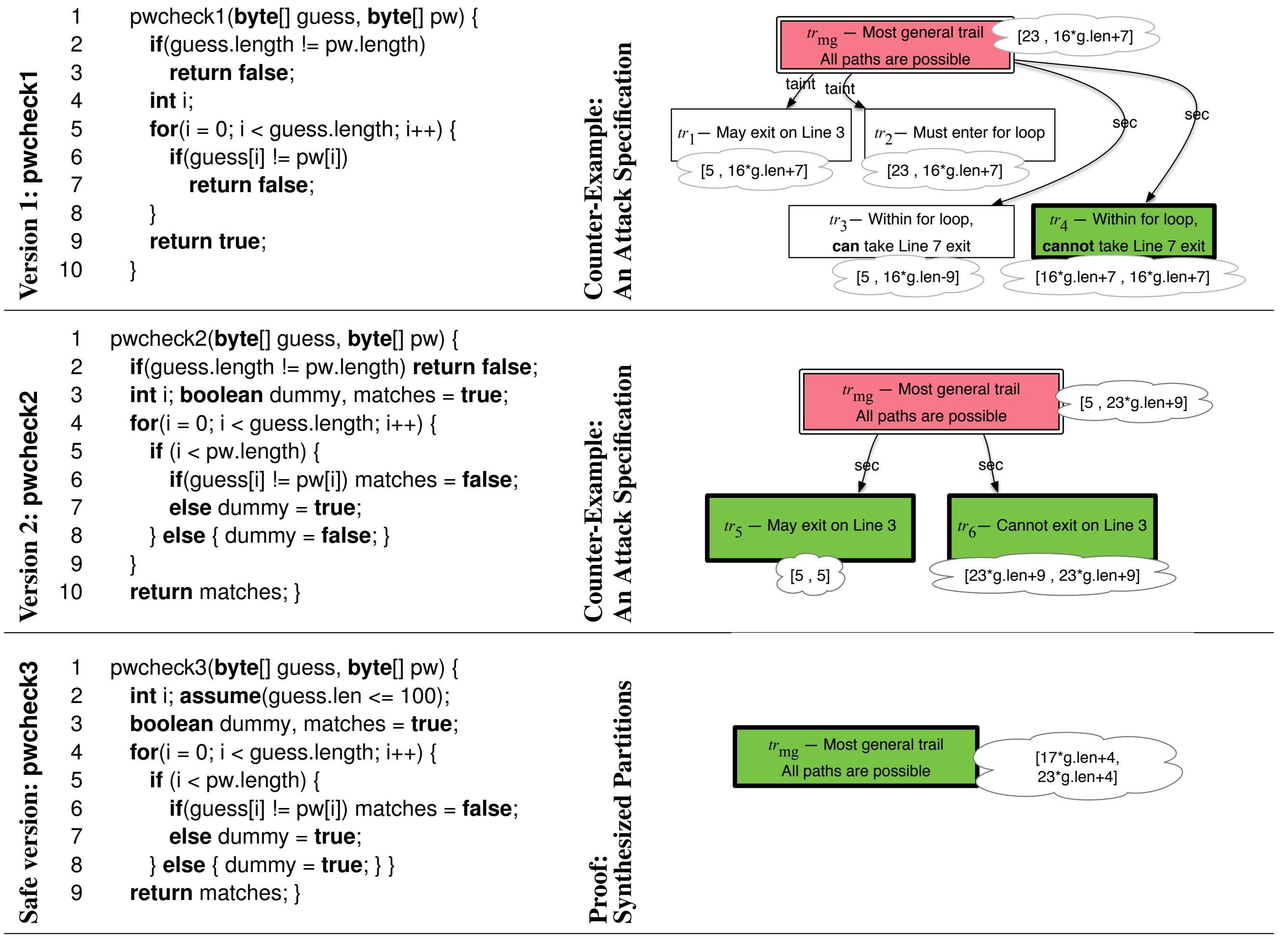
1. The union of all of the trails' languages covers the language of tr_{mg}. That is, $\bigcup_{i\in[1,n]} L(tr_i) \supseteq L(tr_{mg})$.
2. Trails correlate to the branching decisions that depend on low security variables.
3. Each trail tr_i is such that every execution's running time can be described by a single function P_i, *e.g.* $P^{\text{lin}}_{\text{low}}$. 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 (Sec. 3), this suffices to entail that the overall program is free of timing channels. Moreover, when the tool fails to find a partition, it attempts to discover an attack specification.

Running our tool on pwcheck1 takes about 5 seconds, and yields the *tree of trails* displayed on the top right of Fig. 1. A key component of our algorithm is the way that we generate subtrails, embodied in a component called REFINEPARTITION and discussed below. The parent-child relationship in the tree 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). For legibility, we have replaced each trail's regular expression with an intuitive description of the trail it specifies (*Blind review note: the full output can be found in the supplementary material*). Bolded (green) nodes indicate that all trails in its subtree are *safe*; double-outlined (red) nodes indicate an attack specification (multiple subtrails having different running times).

The primary purpose of our tool is to prove safety of the input program. To this end, the tool proceeds using REFINEPARTITION to generate the first two trails (tr_1 and tr_2), splitting executions based on the branching on Line 2 which involves low security (as well as high security) variables.

The next step is, for each subtrail, to determine whether all executions within that subtrail have the same running time, as a function of low security variables. 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 [5] to compute transition invariants [20], and match these invariants against a database of complexity bound lemmas [9, 10]. The result of applying BOUNDANALYSIS to tr_1 is that it computes a lower bound of 5 and an upper bound of $16 \times$ g.len $+ 7$ (depicted as a cloud), where g.len is shorthand for guess.length. Meanwhile, it computes the lower/upper pair $[23, 16 \times \text{g.len} + 7]$ for tr_2.


Version 1: pwcheck1
Counter-Example: An Attack Specification
tr_{mg} — Most general trail All paths are possible
[23 , 16*g.len+7]
taint
taint
sec
sec
tr_1— May exit on Line 3
[5 , 16*g.len+7]
tr_2— Must enter for loop
[23 , 16*g.len+7]
tr_3— Within for loop, can take Line 7 exit
[5 , 16*g.len-9]
tr_4 — Within for loop, cannot take Line 7 exit
[16*g.len+7 , 16*g.len+7]
Version 2: pwcheck2
Counter-Example: An Attack Specification
tr_{mg} — Most general trail All paths are possible
[5 , 23*g.len+9]
sec
sec
tr_5 — May exit on Line 3
[5 , 5]
tr_6— Cannot exit on Line 3
[23*g.len+9 , 23*g.len+9]
Safe version: pwcheck3
Proof: Synthesized Partitions
tr_{mg} — Most general trail All paths are possible
[17*g.len+4, 23*g.len+4]

```
1    pwcheck1(byte[] guess, byte[] pw) {
2      if(guess.length != pw.length)
3        return false;
4      int i;
5      for(i = 0; i < guess.length; i++) {
6        if(guess[i] != pw[i])
7          return false;
8      }
9      return true;
10   }
```

```
1    pwcheck2(byte[] guess, byte[] pw) {
2      if(guess.length != pw.length) return false;
3      int i; boolean dummy, matches = true;
4      for(i = 0; i < guess.length; i++) {
5        if (i < pw.length) {
6          if(guess[i] != pw[i]) matches = false;
7          else dummy = true;
8        } else { dummy = false; }
9      }
10     return matches; }
```

```
1    pwcheck3(byte[] guess, byte[] pw) {
2      int i; assume(guess.len <= 100);
3      boolean dummy, matches = true;
4      for(i = 0; i < guess.length; i++) {
5        if (i < pw.length) {
6          if(guess[i] != pw[i]) matches = false;
7          else dummy = true;
8        } else { dummy = true; } }
9      return matches; }
```

Figure 1. Three 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 secret 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 (g.len is short for guess.length). Bolded (green) nodes indicate areas where the program is safe and double-line (red) nodes indicate an attack specification.

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 can stop subdividing this tr_i and mark it safe (bolded/green). Unfortunately, in this case, the difference between tr_1 and tr_2 is not narrow, and no further sub-partitioning is possible.

In such situations, *Blazer* 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. For tr_4, our toolchain computes a *linear* runtime of $16 \times$g.len$+7$. Meanwhile, tr_3 permits the program to return early either by taking the **return** statement on Line 3 or the **return** statement on Line 7. For tr_3, our toolchain computes a range of runtimes, with a lower bound of constant running time. At this point the tool reports a vulnerability because 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 the lower bound *constant* running time of tr_3 and the *linear* running time of tr_4, 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) [19].

***Try again:* pwcheck2.** With this output, a programmer may take to improving the security of the code. Imagine that they create the new version pwcheck2, eliminating the early **return** in the body of the **for** loop. Running the tool again yields the second output tree in Fig. 1. In this case, tr_5 specifies that executions take the first **return** statement and tr_6 specifies that executions cannot. Again, an attack specification is reported: tr_5 is constant, tr_6 is linear in g.len, and the choice between them depends on high data.

***Correct version:* pwcheck3.** Realizing that this is still a problem, the programmer may create pwcheck3. The tool attempts to compute a complexity bound for tr_{mg} and finds that all executions have a running time in the range of $[17 \times \text{g.len} + 4, 23 \times \text{g.len} + 4]$. Assuming that $\text{g.len} \leq 100$ (line 2), this running time is equivalent to $20 \times \text{g.len} \pm 300$, which is safe assuming the adversary's observational power is bounded by the constant 300.

Type-based approaches. Our work improves over type-based approaches such as [1, 12]. These prior works require that the program (1) not contain a loop with a high-dependent loop condition and (2) not contain a conditional branch with a high-dependent branch condition and the branches having different (statically determined) run times. Here are two trivial examples that cannot be typed with such approaches even though they are secure:

```
void ex1(x,h) { if false { while (h < x) { h++; } }}
void ex2(x,y) {
    if (h > x) { tick; } else { tick;tick;}
    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 7. 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.

Beyond side-channels: Partitions for k-Safety. While we have focused on side channels in this section, our decomposition strategy works for other k-safety properties, too. In the next 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. Formal definitions of trails and our partitioning algorithm follow. We conclude by presenting our tool *Blazer* that is capable of proving side-channel freedom and synthesizing attack specifications for a variety of examples.

3. Semantic partitioning

In this paper we describe a novel approach to proving relational properties using a non-relational analysis. In this section, we formally define the setup and prove why it works. The following sections present our algorithm and experimental results.

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.

Remark 1. 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 $[\![C]\!] = \bigcup_{1 \leq i \leq n} T_i$. 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 for each $T \in \mathfrak{T}$:

$$\begin{aligned}&\forall \pi_1, \ldots, \pi_k \in [\![C]\!]^k .\\ &\psi(\pi_1, \ldots, \pi_k) \Rightarrow (\textstyle\bigwedge_{1 \leq i,j \leq k} \pi_i \in T \Leftrightarrow \pi_j \in T).\end{aligned}$$

Roughly, $\{T_1, \ldots, T_n\}$ is a ψ-quotient partition if for each T_i and each k-tuple of traces satisfying ψ, either all the k traces belong to T_i or none do.

Example 2. Note that, for relational properties (*i.e.*, when $k \geq 2$), the only *true*-quotient partition is the singleton $\{[\![C]\!]\}$. □

Example 3. The partitioning based on low (tainted) inputs described earlier can be formalized as a ψ_{SC}-quotient partition, where $\psi_{SC}(\pi_1, \pi_2)$ 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 $T_>, T_\leq$ described in Section 2 where low is a low variable, $T_>$ contains all traces with low > 0, and $T_\leq$ contains all traces with low ≤ 0. It is easy to see that $\mathfrak{T} = \{T_>, T_\leq\}$ is a ψ_{SC}-quotient partition. □

Let q be a k-safety property below.

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

We say that q is *ψ-quotient partitionable* if

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

In short, to verify or refute a ψ-quotient partitionable it suffices to only consider the k-tuples of execution traces that satisfy ψ.

Example 4. Trivially, any k-safety property is *true*-quotient partitionable. However, as remarked in Example 2, the only possible partition for *true* is the trivial one: $\{[\![C]\!]\}$. $\square$

Example 5. The side-channel attack detection problem described before is the 2-safety property below:

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

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

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 and traces T, denoted $\mathsf{RBPS}(P,q,T)$, if the following holds:

$$(\forall\pi \in [\![C]\!] \cap T.P(\pi)) \Rightarrow\\ \forall\pi_1,\ldots,\pi_k \in ([\![C]\!] \cap T)^k.\Phi_q(\pi_1,\ldots,\pi_k).$$

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

Example 6. Recall the side-channel avoidance property q_{sc} from Example 5. 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 $P_t(\pi)$ for each running time t where $P_t(\pi)$ is true iff the running time of π is close enough to t. It is easy to see that, for any set of traces T, we have $\mathsf{RBPS}(P_t, q_{\mathsf{sc}}, T)$. $\square$

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. *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*

1.) $\mathsf{RBPS}(P,q,T)$; and
2.) 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 1.) and 2.) for T. Then, we have $\Phi_q(\pi_1,\ldots,\pi_k)$ by 2.) and the definition of $\mathsf{RBPS}(P,q,T)$. $\square$

Example 7. Recall the trivial *true*-quotient partition from Example 2. Recall from Example 4 that the only possible partition in this case is $\{[\![C]\!]\}$. 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. $\square$

Example 8. 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 ψ-quotient partition. Such a fine 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 q_{sc}, applying the non-relational complexity analysis described in Example 6 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 $\psi_{SC} \triangleq \mathit{in}(\pi_1)[\boldsymbol{\ell}] = \mathit{in}(\pi_2)[\boldsymbol{\ell}]$ 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. $\square$

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 that we will use to iteratively construct a partition over program traces.

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 as: $tr ::= \epsilon \;||\; b_1b_2 \;||\; tr_1|tr_2 \;||\; tr_1{\cdot}tr_2 \;||\; tr^*$, 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

In this subsection we 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 $tr = tr_1 \;|_l\; tr_2$, 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.

5. 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 an-

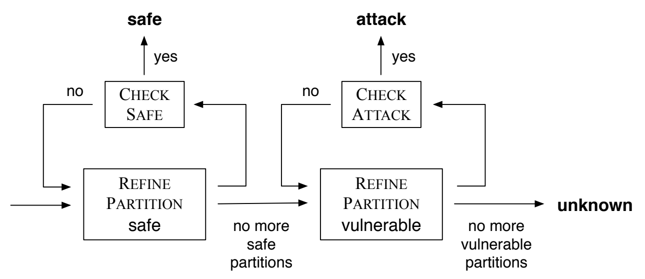


Figure 2. Overall Algorithm

notate 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 6). The main subprocedures of our algorithm are:

- REFINEPARTITION($\mathfrak{T}, \{\mathsf{safe}, \mathsf{vulnerable}\}$): 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 procedure 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.

6. Implementation

We have implemented our technique as the *Blazer* tool for proving the absence of potential timing-channel vulnerabilities in Java program. To process Java bytecode, *Blazer* uses the WALA front-end[3], which transforms bytecode into an SSA-based CFG. We used the information flow (taint) analysis JOANA [21] 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[4] to check language inclusion and construct intersection, union, and complementation automata. We have wrapped this library with a later translation between trails (described as regular expressions) and automata.

We have built a custom abstract interpreter on top of WALA and integrated the Parma Polyhedra Library (PPL) [2], to compute numerical invariants. We have equipped this abstract interpreter with the ability to be restricted to a given trail. We then leverage the seeding technique [5] to compute transition invariants [20], and match these invariants against a database of complexity bound lemmas [9, 10]. This is 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. To this end, we use multiple approaches. First, we use Z3[5] to determine whether two symbolic complexity bounds are the same. Failing that, we have another generic component that computes the highest degree of the complexity bound polynomial, a rough heuristic that works for many examples. In other cases, when a platform-specific

[3] http://wala.sourceforge.net/
[4] http://www.brics.dk/automaton/
[5] https://z3.codeplex.com/

model of execution cost is provided, we use 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 number of instructions. We currently use a simple machine model in which each bytecode instruction is counted as a single unit.

7. Evaluation

We evaluated the implementation of *Blazer* on 25 benchmarks, including 6 examples drawn from the DARPA Space/-Time Analysis for Cybersecurity (STAC) challenge problems[6] and 7 real-world programs in which timing attacks were exploited and reported in cryptography papers [8, 15, 19]. 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.

7.1 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.[7]

- *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. `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.* [8] (*GPT*), Kocher [15] (*K96*), and Pasareanu *et al.* [19] (*PPM16*). The example from *PPM16* is discussed in Section 2 and shown in Fig. 1.

Our implementation of the interprocedural aspect of *Blazer* is incomplete at the time of this writing. For now,

[6] http://www.darpa.mil/program/space-time-analysis-for-cybersecurity

[7] Blind review note: please see the supplementary materials for the complete benchmark sources.

```
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.

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).

Observer modeling. As discussed in the implementation section, observable running time differences are modeled in several ways. For all benchmarks, symbolic running times are first compared using Z3[8] to filter close running times.

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

[8] https://z3.codeplex.com/

Benchmark	Version	Size	Result	Time (s)
MicroBench				
`array`	Safe	16	✓	2.08
`array`	Attack	14	✓	1.19
`login`	Safe	16	✓	1.77
`login`	Attack	11	✓	1.79
`loopAndbranch`	Safe	15	✓	0.84
`loopAndbranch`	Attack	15	✓	3.44
`nosecret`	Safe	7	✓	1.10
`notaint`	Attack	9	✓	6.91
`sanity`	Safe	10	✓	1.53
`sanity`	Attack	9	✓	1.83
`straightline`	Safe	7	✓	0.90
`straightline`	Attack	7	✓	20.16
STAC				
`modPow1`	Safe	18	✓	4.40
`modPow1`	Attack	58	✓	1061.38
`modPow2`	Safe	20	✓	3.25
`modPow2`	Attack	106	†	4792.33
`passwordsEq`	Safe	16	✓	4.09
`passwordsEq`	Attack	15	✓	4.21
Literature				
`k96`	Safe	17	✓	1.66
`k96`	Attack	15	✓	15.11
`gpt14`	Safe	15	✓	2.22
`gpt14`	Attack	26	‡	**T.O.**
`pwcheck1`	Attack	15	✓	5.05
`pwcheck2`	Attack	19	✓	2.74
`pwcheck3`	Safe	15	✓	1.58

Table 1. The results of applying our tool *Blazer* to a variety of benchmarks: hand-made examples, examples from the literature [8, 15, 19], and extracted from DARPA STAC challenge problems. The tool proved safe or provided an attack specification for all but two benchmarks. †`modPow_unsafe` ran out of regex characters while generating subtrails. ‡`gpt14_unsafe` was terminated after taking more than two hours as indicated by T.O. for timeout.

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.

7.2 Results

The benchmarks were run sequentially on a single commodity PC with a quad-core 3.07GHz processor and 12GB of RAM. Table 1 shows the results. The **Benchmark** column identifies the benchmark's method and **Version** identifies whether it's the safe or unsafe version of the method. **Result** indicates whether *Blazer* correctly identifies the method as being safe or having a potential attack with a check-mark. **Size** indicates the number of basic blocks in the method's control-flow graph, and the **Time** column shows the tool's running time in seconds.

Our tool is sound: it either returns a correct result or else gives up (runs out of time/memory). In every Safe benchmark *Blazer* generated a proof of safety and in almost all Attack benchmarks, generated an attack specification. For all Safe benchmarks, *Blazer* is able to prove them to be safe in less than 5 seconds. Perhaps not surprisingly, synthesizing an attack is a more computationally intense task. For the *MicroBench* examples *Blazer* completed typically in a few seconds and synthesized an attack specification for every unsafe micro-benchmark.

The two cases where the tool did not return an answer are the unsafe versions of `modPow2` and `gpt14`. For the former, we ran out of characters in using regular expressions in `brics`, while generating trail automata. For the latter, the tool caused memory thrashing and was terminated after two hours.

The running time of most benchmarks is about five seconds or less. The longer-running benchmarks are related to the number of subtrails and combinations of bound expressions that needed comparison. Pushing our tool, we found that the main bottleneck is in attack detection, because of the high number of combinations of bound expressions to compare in complex benchmarks. We plan to investigate efficient bound comparison techniques in future work.

8. Related work

To the best of our knowledge, our approach is the first method that solves k-safety problems by decomposing the problem and applying non-relational reasoning. We now discuss other strategies to solving k-safety problems.

Self-composition [3, 18, 23, 25] reduces the k-safety problem to a 1-safety problem by composing (sequentially, in parallel, or in an interleaving fashion [18, 23]) k copies of the given program. 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.

Closely related to the self-composition approach is the recent work by Sousa and Dillig [22]. 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 [18, 23], such a lockstep reasoning improves the performance of the approach by more tightly coupling the key invariants across the program copies.

By contrast, the method proposed in this paper does not (explicitly or implicitly) make program copies and reduce the problem to a 1-safety problem, and 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. Next, we discuss more distantly related works.

Static analysis for side-channels. This paper describes a timing-channel attack analysis as an application of the general k-safety property analysis framework. There has been much work on static analysis for detecting and preventing timing attacks. While a complete coverage is not possible, we discuss a few representative works. Agat [1] proposes a type-based program transformation for timing attack prevention that inserts delays to make the program's run time secret independent. Hedin and Sands [12] also propose a type-based approach to detecting timing channel attacks. Zhang *et al.* [26] proposes a framework for preventing timing attacks that addresses low-level issues such as cache uses. It comprises a type-based analysis, hardware assistance, and predictive timing mitigation. Their approach is quantitative in that they ensure boundedness of the amount of information leak. The quantitative approach is also taken in the work by Doychev *et al.* [7] on a static analysis for detecting leaks due to cache uses, and the work by Pasareanu *et al.* [19] that quantifies information leak amount via symbolic execution and model counting.

Our timing attack analysis can be more precise than the type-based approaches. For example, it can handle high-security-dependent conditionals with branches having different run times. But, it does not address low-level issues such as cache, partly due to the fact that such issues are outside of the scope of the DARPA STAC challenge. It also does not address the quantitative aspect of side channels. However, some quantitative bounding problems are known to be k-safety problems [24, 25], and we expect our general analysis framework to be applicable to them. Also, we remark that, due to the under-approximate nature of their technique, Pasareanu *et al.*'s approach [19] is unsound for verifying non-interference (and more generally, quantitative bounds). Nonetheless, we believe this to be a complementary approach: Our tool emits attack specifications that could be used to construct a slice of the program where there is likely to be an attack, and given as input to their tool.

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 [17]. Similarly, PAGAI [14] employs a technique called path focusing [13] to guide an abstract interpreter to consider certain paths at a time. In comparison to control-flow refinement (CFR) [10] 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 k-safety properties, nor permit low-vs-high annotations, nor define a decomposition that reduces k-safety to a trail/path-oriented property.

9. Conclusion

We have presented a novel decomposition technique that reduces k-safety verification to a task fit for a non-relational analysis. In this way, we have departed from prior works that are based on self-composition or some form of state/program replication. We have proved this approach to be sound. Further, toward the issue of timing side-channels, we have introduced a method for symbolically synthesizing path-based partitions (trails) that case-split in instances of taint-based or secret-based branching. We implemented a tool called *Blazer* and demonstrate that it can prove non-interference or synthesize an attack specification on a collection of 25 benchmark examples, including 6 drawn from the STAC challenge problems and 7 from cryptography literature.

We believe that our decomposition sets the stage for the development of more powerful relational analyses and/or other k-safety properties. We are currently improving *Blazer* so that it scales to larger examples. We are also exploring how it can be used to prove relational properties of heap-manipulating programs or relational temporal properties.

References

[1] J. Agat. Transforming out timing leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 40–53, 2000.

[2] 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. *Sci. Comput. Program.*, 72(1-2):3–21, June 2008. ISSN 0167-6423.

[3] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[4] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 200–214, 2011.

[5] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 211–224, 2007.

[6] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[7] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4:1–4:32, June 2015. ISSN 1094-9224.

[8] D. Genkin, I. Pipman, and E. Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731*, pages 242–260, 2014. ISBN 978-3-662-44708-6.

[9] S. Gulwani and F. Zuleger. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 292–304, 2010.

[10] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 375–385, 2009.

[11] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139, 2009.

[12] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1):163–182, 2005.

[13] J. Henry. *Static analysis by path focusing*. PhD thesis, Master's thesis, Grenoble INP, 2011., 2011. http://www-verimag.imag.fr/~{}jhenry/pdf/M2R_report.pdf.

[14] J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.

[15] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

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

[17] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*, pages 5–20. Springer, 2005.

[18] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*, pages 279–296, 2006.

[19] C. S. Pasareanu, Q. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 387–400, 2016.

[20] A. Podelski and A. Rybalchenko. Transition invariants. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 32–41, 2004.

[21] 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): 280–287, 2014.

[22] M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, 2016. ISBN 978-1-4503-4261-2.

[23] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *International Static Analysis Symposium*, pages 352–367. Springer, 2005.

[24] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. *Journal of Computer Security*, 19 (6):1029–1082, 2011.

[25] H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theoretical Computer Science*, 538:167–182, 2014.

[26] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 99–110, 2012.