Hypertesting of Programs: Theoretical Foundation and Automated Test Generation

Michele Pasqua
michele.pasqua@univr.it
Dept. of Computer Science
University of Verona
Verona, Italy

Mariano Ceccato
mariano.ceccato@univr.it
Dept. of Computer Science
University of Verona
Verona, Italy

Paolo Tonella
paolo.tonella@usi.ch
Software Institute
Università della Svizzera italiana
Lugano, Switzerland

ABSTRACT

Hyperproperties are used to define correctness requirements that involve relations between multiple program executions. This allows, for instance, to model *security and concurrency requirements*, which cannot be expressed by means of trace properties.

In this paper, we propose a novel systematic approach for automated testing of hyperproperties. Our contribution is both foundational and practical. On the foundational side, we define a *hypertesting framework*, which includes a novel hypercoverage adequacy criterion designed to guide the synthesis of test cases for hyperproperties. On the practical side, we instantiate such framework by implementing HyperFuzz and HyperEvo, two test generators targeting the *Non-Interference* security requirement, that rely respectively on fuzzing and search algorithms.

Experimental results show that the proposed hypercoverage adequacy criterion correlates with the capability of a hypertest to expose hyperproperty violations and that both HyperFuzz and HyperEvo achieve high hypercoverage and high vulnerability exposure with no false alarms (by construction). While they both outperform the state-of-the-art dynamic taint analysis tool Phosphor, HyperEvo is more effective than HyperFuzz on some benchmark programs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Information flow control*.

KEYWORDS

Search-based testing, Hyperproperties, Information flows, Security testing, Code coverage criteria

ACM Reference Format:
Michele Pasqua, Mariano Ceccato, and Paolo Tonella. 2018. Hypertesting of Programs: Theoretical Foundation and Automated Test Generation. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In addition to functional correctness, software systems are required to respect critical non-functional properties, such as *safety* (e.g., a program should not threaten human life) and *security* (e.g., confidential information should not be leaked publicly). Most existing *software Verification & Validation (V&V)* approaches focus on functional correctness properties that belong to the family of so called *trace properties*. These correspond to program requirements that can be checked by observing single program executions.

Consider, as an example, a smartphone app and suppose to check whether the app crashes or not. If the app can indeed crash, then by observing its execution when inputs are smartly selected or generated, we can eventually notice the crash. There are, however, important program requirements that go beyond the family of trace properties. For instance, *security requirements* may need to compare more than one execution at a time, in order to spot a defect. Consider a smartphone app dealing with users' confidential information that must not be leaked to public channels (e.g., unsecured connections). In order to *precisely* spot an information leakage, observing individual executions in isolation is not sufficient: we need to compare *pairs* of executions. Indeed, checking single executions may lead to *false alarms*, even in the case of dynamic techniques that are usually supposed to be precise. For instance, *dynamic taint analysis* may yield false alarms when running a single execution and propagating taint tags that reach, but do not have any effect on, public outputs. To precisely check information leaks, we have to execute the app at least twice, with different confidential information values, to assess whether these executions may generate public outputs that also differ, hence leaking information (i.e., in a secure program, when *only* confidential information changes, public information must remain the same).

Other requirements that cannot be expressed as trace properties include functional and safety properties of *concurrent systems*, where multiple parallel executions must satisfy atomicity of some operations or freedom from deadlock. Other examples comprise: security of cryptographic protocols [34]; planning in multi-agents systems [5]; robustness of robotic control mechanisms [36]; code obfuscation [25] and certified compilation techniques [33].

Collectively, those complex requirements that involve more than one execution trace have been named *hyperproperties* [11]. Formally, while trace properties are defined in terms of sets of executions that satisfy a given correctness requirement, hyperproperties are defined in terms of *sets of sets* of program executions, with a specific hyperproperty being the set of all sets of executions (i.e., program semantics) that satisfy the associated requirement. In other words, a hyperproperty stipulates a property over the program semantics

and not a property over individual program executions (as trace properties). This added level of complexity allows to specify *relations* between multiple, different executions of a program, that are not expressible with simple trace properties. Not surprisingly, the gained expressiveness requires more complex V&V techniques.

Current state-of-the-art V&V approaches mainly deal with trace properties, thus a lot of crucial program correctness requirements are not properly considered. Some progress in such direction has been achieved by using static analysis techniques (e.g., based on abstract interpretation [2]), but these solutions do not scale to large and complex software systems. Moreover, such analyses return a sound, conservative superset of the possible violations, which often includes a large proportion of false positives. Hence, the alternative of exposing hyperproperty violations by means of dynamic analysis techniques (e.g., automated test case generation) is extremely appealing [21].

In this paper, we aim at answering the following question:

"How can we define a systematic framework for the dynamic verification of hyperproperties and for the automated generation of execution traces that violate them?"

We first tackle the problem from a theoretical point of view, by developing a foundational theory for the systematic definition of testing strategies suitable for hyperproperties verification: the *hypertesting framework*, which includes a novel *hypercoverage* adequacy criterion as cornerstone. Second, we propose two input generation approaches specifically targeting hyperproperties: one based on fuzzing, dubbed *Hyper-Fuzzing*, and another based on evolutionary search algorithms, dubbed *Evolutionary Hypertesting*. These two approaches are respectively implemented in two tools, HyperFuzz and HyperEvo, which instantiate the proposed hypertesting framework to test a specific security hyperproperty, *Non-Interference* (i.e., absence of leakage of confidential information). We validated the two tools on a state-of-the-art benchmark for security containing vulnerable and non-vulnerable Java programs. Results show that our approach is very accurate in detecting hyperproperty violations, outperforming the state-of-the-art taint analysis tool Phosphor [4] (that can be used to *approximate* Non-Interference).

Summary of Contributions. The paper contributes with the following theoretical and practical results:

- a *theoretical framework* for the systematic testing of hyperproperties, comprising novel *coverage criterion* and structural *search metaheuristics* specific for hyperproperties;
- two *test input generation* approaches for hyperproperties, one based on *fuzzing* and another based on *search algorithms*, together with novel *crossover* and *mutation* operators specific for hyperproprties; and
- two *automated tools*, HyperFuzz and HyperEvo, customized to test a given security hyperproperty (Non-Interference).

Synopsis. In Section 2 we present the proposed hypertesting framework. Then, in Section 3, we describe the hypertesting procedure, declined in two variants: fuzzing-based and search-based. The empirical evaluation of the approach is reported in Section 4, together with the collected results. In Section 5 we compare our approach with the related work. Finally, in Section 6 we draw conclusions and discuss future research directions.

2 HYPERTESTING FRAMEWORK

Given a set of values $\mathbb{V}$ and a set of variables $\mathbb{X}$, a *variables assignment* (often called memory, store or state) is a function $\mathbb{m} \in \mathbb{M} \triangleq \mathbb{X} \rightarrow \mathbb{V} \cup \{\bot\}$ mapping variables to values (here $\bot$ denotes an undefined value). In the following, we model program executions $e \in \mathbb{E}$ as finite sequences of memories, namely $\mathbb{E} \triangleq \bigcup_{n \in \mathbb{N}} \mathbb{M}^n$. Given a program P, we denote with $\text{inputVars}(P) \subseteq \mathbb{X}$ (resp. $\text{outputVars}(P) \subseteq \mathbb{X}$) the set of its input (resp. output) variables. In this setting, an input (resp. output) for P is a variables assignment $\mathbb{m}$ that is defined for all input (resp. output) variables of P, namely such that $\text{dom}(\mathbb{m}) = \text{inputVars}(P)$ (resp. $\text{dom}(\mathbb{m}) = \text{outputVars}(P)$), where $\text{dom}(\mathbb{m})$ is the set of all variables $\mathtt{x}$ for which $\mathbb{m}(\mathtt{x}) \neq \bot$ (the domain of $\mathbb{m}$). Given an execution $e \in \mathbb{E}$ of P, we denote with $e^i \in \mathbb{M}$ (resp. $e^o \in \mathbb{M}$) the input (resp. output) of e. This means that program P produces the program execution e when running starting from the (input) memory e^i, yielding the (output) memory e^o.

The *Control Flow Graph (CFG)* of a program is an abstract representation of the program semantics (i.e., of all program executions), embedding control and data flow information of program variables. The nodes of such graph (usually called *basic blocks*) represent the statements of the program, while arcs represent an execution flow between statements. For instance, in Figure 1 (on the right) we have the CFG of a simple program code snippet (on the left). As we can see from the figure, each statement of the program is labeled with a unique *program point* (the underlined reddish number on the left of the statement) and sequential statements (i.e., sequences of assignments that are not interleaved by conditionals) are grouped in multi-statement blocks. The program state at each program point is the one computed *after* the execution of the command pointed by such program point. In a CFG special blocks are added: an *entry point* (**entry**), indicating the beginning of the program (with typical label ⓪); and an *exit point* (**exit**), indicating the end of the program (with typical label ⓔ). We assume, w.l.o.g., that each program has unique entry and exit points.

Hyperproperties are in general complex, usually modeled by using very expressive logical systems [10]. In this paper we focus on a particular subset of hyperproperties, the *relational k-bounded* [23], that are sufficient to express lots of security and concurrency requirements. Relational here means that the hyperproperty stipulates a relation between the input and the output of a program, while k-bounded means that the executions needed to refute the hyperproperty can be limited to a fixed finite number k. A k-bounded hyperproperty, *k-hyperproperty* for short, is hence of the form:

$$\forall e_1 \in \mathbb{E} \ldots \forall e_k \in \mathbb{E} \,.\, \mathbb{P}_i(e_1^i, \ldots, e_k^i) \Rightarrow \mathbb{P}_o(e_1^o, \ldots, e_k^o) \qquad (1)$$

such that $\mathbb{P}_i \subseteq \mathbb{M}^k$ is a k-ary predicate on inputs and $\mathbb{P}_o \subseteq \mathbb{M}^k$ is a k-ary predicate on outputs[1]. Note that, being $e_1, \ldots, e_k \in \mathbb{E}$, we have that, for instance, e_1 is a program execution, modeled as a sequence of program memories, where e_1^i is the input (memory) while e_1^o is the output (memory). Given a program P, the input predicate $\mathbb{P}_i$ and the output predicate $\mathbb{P}_o$ of Equation (1) are defined for P as predicates on P's variables at the entry and the exit point of P, respectively.

[1]The notation $\mathbb{P}(\mathbb{m}_1, \ldots, \mathbb{m}_k)$, with $\mathbb{P} \subseteq \mathbb{M}^k$, is a shorthand for $(\mathbb{m}_1, \ldots, \mathbb{m}_k) \in \mathbb{P}$.

Snippet of code:

```
if (key == 0) {
  log = key + 5;
} else {
  if (log == 0) {
     log = 5;
  } else {
     if (key > 6) {
        log = key;
     } else {
        log = 0;
        key = 1;
     }
  }
}
```

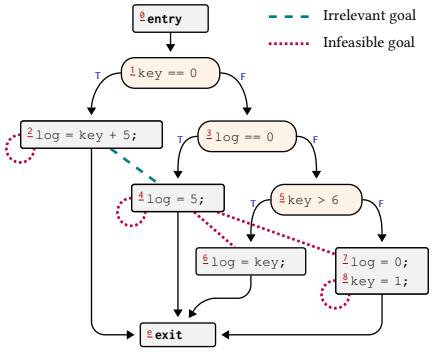

Figure 1: A snippet of code and the corresponding CFG.

For instance, the classic notion of *Non-Interference* [12], that requires the lack of (semantic) dependency on confidential information by public/non-confidential resources, is defined as:

$$\forall e_1 \in \mathbb{E}\, \forall e_2 \in \mathbb{E}\,.\, =_{\mathsf{L}}(e_1^i, e_2^i) \Rightarrow =_{\mathsf{L}}(e_1^o, e_2^o) \tag{2}$$

where $=_{\mathsf{L}}$ says that two memories agree on the values of L (i.e., public) variables[2]. The idea is that there exists a (possibly harmful) *information flow* from a (possibly confidential) variable `x` to a (possibly public) variable `y` in a program P whenever a change in `x` is conveyed to `y` by the execution of P. Equation (2) models the absence of (potentially harmful) information flows from confidential to public variables.

In Figure 1 (on the left) we have a simple snippet of Java-like code that does not satisfy Equation (2), when `key` is considered confidential and `log` is considered public. Indeed, the predicate $=_{\mathsf{L}}$ checks the value of public, i.e., L, variables (`log` in this case) at the beginning ot the program (before the first line) and at the end of the program (after last line). It is easy to find two executions that violate the hyperproperty. For instance, take two executions that have the following inputs: one having `key` equal to 0 and another having `key` equal to 1, with both executions having `log` equal to 6. If we execute the program on those inputs, we obtain the value 5 for `log` in one case and the value 1 in the other. Thus, Non-Interference is violated, since L-equivalent input variables are mapped to non L-equivalent output variables.

Syntactic dependencies, exploited by state-of-the-art approaches to track information flows (e.g., taint analysis), provide an *approximation* of Non-Interference, that models semantic dependencies. This is very often (implicitly) done by existing V&V approaches: the verification of a hyperproperty is approximated by considering a (simpler) trace properties [23]. Being an approximation, it may of course lead to false positives (even in the case of dynamic approaches, that are usually supposed to be precise). Imagine to remove the program points 4, 5, 6, 7 and 8 from the CFG of Figure 1, and to replace the conditional at 3 with the assignment `log == 5;`. The resulting program is secure since, independently from the initial value of `key`, the final value of `log` is always 5. Nevertheless, we have a syntactic dependency between `key` and `log`, potentially inducing a taint analysis to fire a (false) alarm.

[2]A mapping between variables and security levels (L, meaning public, and H, meaning confidential) is given for a program. Flows from H to L variables are not allowed.

To be refuted, a k-hyperproperty requires k properly chosen program executions, hence a *testcase* for a k-hyperproperty has necessarily to model k program executions. For this reason, we model a *test input* for a k-hyperproperty as a tuple of (input) memories.

Definition 1 (Hypertest Input). *Given a k-hyperproperty* hp*, with input predicate $\mathbb{P}_i$, and a program P, a* hypertest input *for* hp *and P is a tuple $(\mathbb{m}_1, \ldots, \mathbb{m}_k)$ such that:*

- $\mathrm{dom}(\mathbb{m}_i) = \mathrm{inputVars}(P)$*, for all $i \in [1..k]$; and*
- $\mathbb{P}_i(\mathbb{m}_1, \ldots, \mathbb{m}_k)$.

To assess whether a testcase is successful or not, we need a notion of *oracle*. In the case of hyperproperties, the oracle would check the satisfaction of the output predicate.

Definition 2 (Hyperoracle). *Given a k-hyperproperty* hp*, with output predicate $\mathbb{P}_o$, a* hyperoracle *for* hp *is a function $O^{\mathsf{hp}} \in \mathbb{E}^k \rightarrow \{0, 1\}$ defined as:*

$$O^{\mathsf{hp}}(e_1, \ldots, e_k) = 1 \;\; \textit{iff} \;\; \mathbb{P}_o(e_1^o, \ldots, e_k^o)$$

We assume O^{hp} computable (and, hence, $\mathbb{P}_o$ decidable), meaning that we can always define a terminating program able to assess whether a testcase is successful or not.

Definition 3 (Hypertest Suite). *Given a k-hyperproperty* hp *and a program P, a* hypertest suite *for* hp *and P is a pair (O^{hp}, I), with $I \subseteq \mathbb{M}^k$, such that:*

- *O^{hp} is a computable hyperoracle for* hp *(Definition 2);*
- *I is a finite non-empty set; and*
- *for all $(\mathbb{m}_1, \ldots, \mathbb{m}_k) \in I$, $(\mathbb{m}_1, \ldots, \mathbb{m}_k)$ is a hypertest input for* hp *and P (Definition 1).*

2.1 Adequacy Criteria for Hyperproperties

Although a k-hyperproperty involves universal quantification over all executions (see Equation 1), to refute a k-hyperproperty it is enough to find a subset of executions that can potentially violate it. For instance, if we consider Non-Interference (a k-hyperproperty having $k = 2$) in the program of Figure 1, the pairs of program points with assignment to the L variable `log` which may potentially produce different L output values when H variables change, are a subset of the cartesian product of all program points where `log` is assigned, namely $\{2, 4, 6, 7\} \times \{2, 4, 6, 7\}$ (16 pairs in total). The cartesian product is justified by the fact that for Non-Interference pairs of executions suffice to refute such hyperproperty (being $k = 2$). Among them, only four pairs of program points are actually worth checking: $(2, 6)$, $(2, 7)$, $(6, 6)$ and $(6, 7)$. In fact, the pair $(2, 4)$ is useless, because `log` is assigned the constant value 5 in all possible executions reaching the program points 2 and 4. For the same reason, the auto-pairs $(2, 2)$, $(4, 4)$ and $(7, 7)$ are useless, because again the same constant value is always assigned to `log` in all paired executions. As the same value 5 is assigned at 2 and 4, we can deem these two program points as equivalent for the purpose of Non-Interference confutation. Finally, as Non-Interference has a symmetric input predicate $\mathbb{P}_i$, given a pair (ℓ, ℓ'), we do not need to consider its inverse (ℓ', ℓ). This leaves us with four pairs of program points that assign `log` to be covered by (paired) executions in order to refute Non-Interference for the example in Figure 1. While covering even such smaller subset of the pairs of program points

that assign `log` for all possible input values remains infeasible, we can use *coverage* of these interesting assignment pairs as an adequacy criterion for the generation of hypertest inputs.

For a given k-bounded hyperproperty hp, we define the *interesting* tuples of executions as the set

$$\mathsf{Core}^{\mathsf{hp}} \triangleq \left\{(e_1,\ldots,e_k) \in \mathbb{E}^k \;\middle|\; \mathbb{P}_i(e_1^i,\ldots,e_k^i) \wedge \neg\mathbb{P}_o(e_1^o,\ldots,e_k^o)\right\}$$

that is, the set of all counterexample executions (i.e., hyperproperty violations) for hp.

We can define a coverage criterion for a given k-hyperproperty as follows. A variable definition is the program point of an assignment where the variable is on the left-hand side. We denote with $\mathsf{defs}(\mathcal{G},\mathtt{x})$ the set of all definitions of the variable `x` in the CFG $\mathcal{G}$. For instance, let $\mathcal{G}^{\mathsf{ni}}$ be the CFG of the program in Figure 1, we have $\mathsf{defs}(\mathcal{G}^{\mathsf{ni}},\mathtt{key}) = \{\underline{8}\}$ and $\mathsf{defs}(\mathcal{G}^{\mathsf{ni}},\mathtt{log}) = \{\underline{2},\underline{4},\underline{6},\underline{7}\}$.

Given a predicate $\mathbb{P}$, we denote with $\mathsf{vars}(\mathbb{P})$ the *scope* of $\mathbb{P}$, that is, the set of variables on which $\mathbb{P}$ acts on. Given a variable `x` and a program point $\underline{\ell}$ in the CFG $\mathcal{G}$, we denote with $\mathsf{vals}(\mathtt{x},\underline{\ell},\mathcal{G})$ the set of all possible values `x` may take at $\underline{\ell}$ in $\mathcal{G}$. Again referring to the program in Figure 1, we have that $\mathsf{vals}(\mathtt{log},\underline{4},\mathcal{G}^{\mathsf{ni}}) = \{5\}$ and $\mathsf{vals}(\mathtt{log},\underline{6},\mathcal{G}^{\mathsf{ni}}) = \{n \in \mathbb{Z} \setminus \{0\} \mid n > 6\}$.

Given a set of program points $\{\underline{\ell_1},\ldots,\underline{\ell_n}\}$ of a CFG $\mathcal{G}$, we denote with $\mathsf{dataSlice}(\mathcal{G},\{\underline{\ell_1},\ldots,\underline{\ell_n}\},d)$ its d-bounded *data-slice* [15] in $\mathcal{G}$, that is the set of all program points in $\mathcal{G}$ having a (transitive) data dependency of depth at most d on statements at program points $\underline{\ell_1},\ldots,\underline{\ell_n}$. A program point $\underline{\ell'}$ is data dependent on program point $\underline{\ell}$ when the instruction at $\underline{\ell'}$ uses a variable `x` defined at $\underline{\ell}$ and a path exists in the CFG between $\underline{\ell}$ and $\underline{\ell'}$ containing no definition of `x`. A bounded data-slice transitively computes data dependencies in the backward direction, starting from the target program points, up-to a given threshold d.

Definition 4 (Value-Insensitive Hypercoverage Goal). *Given a k-hyperproperty* hp*, with output predicate $\mathbb{P}_o$, and a program P, a* value-insensitive hypercoverage goal *for* hp *and P is a tuple* $(\mathtt{x},\underline{\ell_1},\ldots,\underline{\ell_k})$*, where $\underline{\ell_1},\ldots,\underline{\ell_k}$ are not necessarily different, such that:*

- *$\underline{\ell_1},\ldots,\underline{\ell_k}$ are program points in the CFG $\mathcal{G}^P$ of P; and*
- *$\underline{\ell_1},\ldots,\underline{\ell_k} \in \mathsf{defs}(\mathcal{G}^P,\mathtt{x}) \cup \mathsf{dataSlice}(\mathcal{G}^P,\mathsf{defs}(\mathcal{G}^P,\mathtt{x}),d) \cup \{\underline{e}\}$, such that $\mathtt{x} \in \mathsf{vars}(\mathbb{P}_o)$ and $d \geq 1$.*

Consider again the program in Figure 1. Variable `log` is in the scope of the Non-Interference output predicate $=_{\mathsf{L}}$, and the program points $\underline{6}$ and $\underline{7}$ are definitions of `log` in $\mathcal{G}^{\mathsf{ni}}$. Hence, $(\mathtt{log},\underline{6},\underline{7})$ is a value-insensitive hypercoverage goal.

Given a k-hyperproperty hp, with output predicate $\mathbb{P}_o$, and a program P, we define the *value-insensitive hypercoverage criterion* for hp and P, written $\mathsf{VIHCC}(\mathsf{hp},P)$, as the set of all value-insensitive hypercoverage goals for hp and P. The idea behind *hypercoverage* is that coverage of all k-tuples of output variable definitions observed in k different executions ensures that no definition potentially leading to a violation of the output predicate is left untried. On the other hand, covering all k-tuples of output variable definitions with all possible values is unaffordable from a computational point of view, as it represents a form of exhaustive testing, hence theoretically intractable as computing all possible variable values at a program point is undecidable. For these reasons, we rely on a *value-insensitive* hypercoverage criterion, providing an over-approximation of all possible violations (but effectively computable). Indeed, the value-insensitive hypercoverage criterion provides a necessary, but not sufficient, condition to expose a hyperproperty violation[3].

```
procedure ValueInsensitiveCriterion(cfg, ℙ_i, ℙ_o, k, ω)
1   inVars ← VarsOf(ℙ_i)
2   outVars ← VarsOf(ℙ_o)
3   goals ← ∅
4   for x in outVars do
5       defs ← DefinitionsOf(cfg, x)
6       deps ← BoundedDataSlice(cfg, defs, ω)
7       locs ← defs ∪ deps ∪ {e}
8       for (ℓ_1, ..., ℓ_k) in Combinations(locs, k) do
9           goals ← goals ∪ {(x, ℓ_1, ..., ℓ_k)}
        end
    end
10  fwSlice ← ForwardSlicing(cfg, ⟨0, inVars⟩)
11  constVars ← ConstantPropagation(cfg)
12  redundantGoals ← ∅
13  for (x, ℓ_1, ..., ℓ_k) in goals do
14      if {ℓ_1, ..., ℓ_k} ⊈ DefinitionsOf(fwSlice, x) or x ∈ constVars then
15          redundantGoals ← redundantGoals ∪ {(x, ℓ_1, ..., ℓ_k)}
        end
    end
16  return goals \ redundantGoals
```

Algorithm 1: Computing hypercoverage goals.

Proposition 1. *If P violates the k-hyperproperty* hp*, the k executions $(e_1,\ldots,e_k) \in \mathsf{Core}^{\mathsf{hp}}$ that witness the violation cover one hypercoverage goal in $\mathsf{VIHCC}(\mathsf{hp},P)$.*

Among the value-insensitive hypercoverage goals, there are *redundant* elements, namely goals that may be ignored without affecting the possibility of counterexample generation. Dropping such elements will produce a narrower search space, hence improving the testing performance. In other words, redundant goals represent execution tuples that do not belong to $\mathsf{Core}^{\mathsf{hp}}$, hence they do not concur to the falsification of the hyperproperty. We can identify two sources of redundancy: infeasible goals and irrelevant goals.

Infeasible goals represent program point tuples that cannot be simultaneously covered by execution tuples in $\mathsf{Core}^{\mathsf{hp}}$. Starting from executions $e_1,\ldots,e_k$ such that $\mathbb{P}_i(e_1^i,\ldots,e_k^i)$, the program points in infeasible goals are not exercised in any of the possible k executions satisfying $\mathbb{P}_i$. In Figure 1, infeasible program point pairs are linked by red dotted lines (e.g., the goal $(\mathtt{log},\underline{4},\underline{7})$ is infeasible).

Definition 5 (Infeasible Goal). *Given a k-hyperproperty* hp*, with input predicate $\mathbb{P}_i$, and a program P, a goal $(\mathtt{x},\underline{\ell_1},\ldots,\underline{\ell_k}) \in \mathsf{VIHCC}(\mathsf{hp},P)$ is said* infeasible *when for all executions $e_1,\ldots,e_k$ of P such that $\mathbb{P}_i(e_1^i,\ldots,e_k^i)$ holds, we have that $\underline{\ell_1},\ldots,\underline{\ell_k}$ are not* all *reachable in $e_1,\ldots,e_k$.*

Irrelevant goals represent program point tuples having definitions that do not refute the hyperproperty. This can be seen as an output-driven selection: executions $e_1,\ldots,e_k$ such that $\mathbb{P}_o(e_1^o,\ldots,e_k^o)$ can be ignored, since they do not provide a counterexample for the hyperproperty. In Figure 1, irrelevant program point pairs are linked by green dashed lines (e.g., the goal $(\mathtt{log},\underline{2},\underline{4})$ is irrelevant).

[3] Proof omitted due to lack of space.

Definition 6 (Irrelevant Goal). *Given a k-hyperproperty* hp, *with output predicate $\mathbb{P}_o$, and a program P with CFG $\mathcal{G}^P$, a goal $(\mathtt{x}, \underline{\ell_1}, \ldots, \underline{\ell_k}) \in \mathsf{VIHCC}(\mathsf{hp}, P)$ is said* irrelevant *when for all values $v_1 \in \mathsf{vals}(\mathtt{x}, \underline{\ell_1}, \mathcal{G}^P), \ldots, v_k \in \mathsf{vals}(\mathtt{x}, \underline{\ell_k}, \mathcal{G}^P)$ we have $\mathbb{P}_o(v_1, \ldots, v_k)$.*

Efficient Hypercoverage Goals Computation. Infeasible and irrelevant goals still involve semantic aspects of a program, hence, we cannot precisely compute such elements. However, we can settle for approximations of such sets that are efficiently computable.

In particular, we can approximate infeasible goals by computing a *forward slice* [6] of the program, using $\langle \underline{0}, \mathsf{vars}(\mathbb{P}_i) \rangle$, where $\underline{0}$ is the entry point of the program, as slicing criterion. This means that we cover only definitions affected by input variables or by decisions that depend on input variables.

We can approximate irrelevant goals by performing a *constant propagation* [37] analysis of the program and excluding all program points where output variables are constant. This means that we cover only definitions that can potentially lead to different values for the output variables. This is indeed a very coarse approximation, since we need such variables to be constant only when k paths satisfying the considered hypercoverage goal are traversed, not in the execution of k arbitrary paths. We plan to find a more clever strategy to prune irrelevant goals as a future work.

Algorithm 1 computes the value-insensitive hypercoverage criterion (lines **3** – **9**). Then, irrelevant and infeasible goals, if any, are removed. For the latter purpose, we compute a forward slice (line **10**) and a constant propagation (line **11**) of the program, in order to detect which goals are redundant (lines **12** – **15**). This is done by either checking that the program points of a goal are not in the slice, or the variable of a goal is constant (line **14**). Finally, the redundant goals found are removed from the value-insensitive hypercoverage criterion (line **16**).

Distance Metrics for Hyperproperties. A distance metric for hyperproperties measures the distance between the execution traces of a candidate k-tuple of tests and a target, yet uncovered, hypertesting coverage goal. This distance is used to guide test generation, as explained in the next section.

To define a distance metric for hyperproperties, we use standard structural search metaheuristics, such as *approach level* and *branch distance* [26], adapted to our setting. In particular, given a memory $\mathbb{m}$ and a program point $\underline{\ell}$, belonging to an implicitly referenced program P, we denote with: $\mathsf{AL}(\underline{\ell}, \mathbb{m})$, the minimum number of control nodes between a statement executed by P on $\mathbb{m}$ and the statement at $\underline{\ell}$ (approach level); and $\mathsf{BD}(\underline{\ell}, \mathbb{m})$, the distance, computed according to any branch computation scheme [26], between the variable values involved in a *condition* whose truth value makes $\underline{\ell}$ unreachable in the given execution and those achieving the opposite truth value, which would make $\underline{\ell}$ reachable after execution of the given *condition*. The branch distance is 0 if the statement at $\underline{\ell}$ has been reached by executing P on $\mathbb{m}$. Here, the considered *condition* is the boolean expression corresponding to the closest (w.r.t. the statement at $\underline{\ell}$) control node in a path not leading to the statement at $\underline{\ell}$. The *single-run distance* of the statement at $\underline{\ell}$ w.r.t. the input (memory) $\mathbb{m}$, denoted by $\mathsf{SRD}(\underline{\ell}, \mathbb{m})$, is defined as $\mathsf{AL}(\underline{\ell}, \mathbb{m}) + \mathsf{BD}(\underline{\ell}, \mathbb{m})$.

Approach level and branch distance are the basic components of the multi-run distance needed for hyperproperties. Indeed, to satisfy a hypercoverage goal we have to cover k program points in the k executions associated with a hypertest, which consists of k inputs. Hence, we need a distance metric considering k execution paths, not just one. In Figure 2, we consider the case of $k = 2$. In the picture, we aim at covering two program points, $\underline{\ell_1}$ and $\underline{\ell_2}$, by performing two program executions, one yielding from $\mathbb{m}_1$ and another yielding from $\mathbb{m}_2$. As we can see, the goal $(\underline{\ell_1}, \underline{\ell_2})$ is not covered, since we do not encounter $\underline{\ell_1}$ and $\underline{\ell_2}$ along the paths yielding from $\mathbb{m}_1$ and $\mathbb{m}_2$. To measure how much we are far from the goal, we compute the single-run distances of each execution from each program point and we take the minimum of the their sum. Of course, one execution can cover only one program point, hence the only interesting combinations are those with different control points and different input memories, namely $\mathsf{SRD}(\underline{\ell_1}, \mathbb{m}_1)$ paired with $\mathsf{SRD}(\underline{\ell_2}, \mathbb{m}_2)$ and $\mathsf{SRD}(\underline{\ell_1}, \mathbb{m}_2)$ paired with $\mathsf{SRD}(\underline{\ell_2}, \mathbb{m}_1)$. In the example, we considered the case of $k = 2$ for the sake of simplicity, but the definition can be given for an arbitrary $k > 1$.

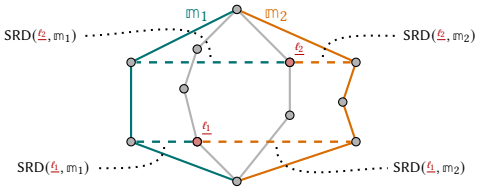

Figure 2: Graphical explanation of the Multi-Run Distance.

Definition 7 (Multi-Run Distance). *Given a hypercoverage goal $g = (\mathtt{x}, \underline{\ell_1}, \ldots, \underline{\ell_k})$ and a hypertest input $t = (\mathbb{m}_1, \ldots, \mathbb{m}_k)$, we define the* multi-run distance *of g w.r.t. t, written $\mathsf{MRD}(g, t)$, as*

$$\mathsf{MRD}(g,t) \triangleq \min \left\{ \begin{array}{l} \sum_{i=1}^{k} \mathsf{SRD}(\underline{\ell_i}, \mathbb{m}_{j_i}) + \\ \mathsf{P}_{\mathtt{x}}(\overset{\ell_1}{\underline{\ell_1}}[\![\mathtt{x}]\!]\mathbb{m}_{j_1}, \ldots, \overset{\ell_k}{\underline{\ell_k}}[\![\mathtt{x}]\!]\mathbb{m}_{j_k}) \end{array} \middle| \; j_1 \neq \ldots \neq j_k \right\}$$

where $\overset{\ell}{\underline{\ell}}[\![\mathtt{x}]\!]\mathbb{m}$ is the value of $\mathtt{x}$ at program point $\underline{\ell}$ obtained by executing the program on $\mathbb{m}$.

As a matter of example, when $k = 2$, hence with $g = (\mathtt{x}, \underline{\ell_1}, \underline{\ell_2})$ and $t = (\mathbb{m}_1, \mathbb{m}_2)$, such definition becomes $\mathsf{MRD}(g, t) \triangleq$

$$\min \left\{ \begin{array}{l} \mathsf{SRD}(\underline{\ell_1}, \mathbb{m}_1) + \mathsf{SRD}(\underline{\ell_2}, \mathbb{m}_2) + \mathsf{P}_{\mathtt{x}}(\overset{\ell_1}{\underline{\ell_1}}[\![\mathtt{x}]\!]\mathbb{m}_1, \overset{\ell_2}{\underline{\ell_2}}[\![\mathtt{x}]\!]\mathbb{m}_2), \\ \mathsf{SRD}(\underline{\ell_1}, \mathbb{m}_2) + \mathsf{SRD}(\underline{\ell_2}, \mathbb{m}_1) + \mathsf{P}_{\mathtt{x}}(\overset{\ell_1}{\underline{\ell_1}}[\![\mathtt{x}]\!]\mathbb{m}_2, \overset{\ell_2}{\underline{\ell_2}}[\![\mathtt{x}]\!]\mathbb{m}_1) \end{array} \right\}$$

In Definition 7 we insert a *penalty* component $\mathsf{P}_{\mathtt{x}}(v_1, \ldots, v_k)$ to give more importance to execution pairs that yield values for $\mathtt{x}$ falsifying the output predicate $\mathbb{P}_o$. The penalty is defined as $\mathsf{P}_{\mathtt{x}}(v_1, \ldots, v_k) \triangleq \epsilon\, \delta_{\mathtt{x}}(v_1, \ldots, v_k)$, where ϵ is an arbitrary small constant, while $\delta_{\mathtt{x}} \in \mathbb{V}^k \to \{0, 1\}$ is a function defined as:

$$\delta_{\mathtt{x}}(v_1, \ldots, v_k) = 1 \;\; \textit{iff} \;\; \mathbb{P}_o(\mathbb{m}_1[^{v_1}/_{\mathtt{x}}], \ldots, \mathbb{m}_1[^{v_k}/_{\mathtt{x}}])$$

for some $\mathbb{m}_1, \ldots, \mathbb{m}_k$ such that $\mathbb{P}_o(\mathbb{m}_1, \ldots, \mathbb{m}_k)$.

As a matter of example, in the case of $k = 2$ and taking $\mathbb{P}_o$ as the equality relation for the variable $\mathtt{x}$, the penalty becomes $\mathsf{P}_{\mathtt{x}}(v_1, v_2) = \epsilon\, \delta_{\mathtt{x}}(v_1, v_2)$, where: $\delta_{\mathtt{x}}(v_1, v_2) = 1$, if $v_1 = v_2$; and $\delta_{\mathtt{x}}(v_1, v_2) = 0$, otherwise.

3 HYPERTEST INPUT GENERATION

To effectively test a k-hyperproperty, by instantiating the framework proposed in the previous section, we have to: *(i)* generate

hypertest inputs that satisfy the hyperproperty input predicate; *(ii)* run the k executions for each hypertest input; and *(iii)* check the satisfaction of the hyperproperty output predicate. If check *(iii)* fails, we have a hyperproperty violation, and the corresponding hypertest input provides the counterexample.

To assess how much we have tested a program, we can exploit the value-insensitive hypercoverage criterion. Indeed, if we cover a large portion of hypercoverage goals but no violation is found, we may say with confidence that the program is likely to satisfy the hyperproperty. Such *hypertesting procedure* is summarized in Algorithm 2, where `HyperTester` is a suitable strategy to craft hypertest inputs. We provide two of such strategies in the following.

3.1 Fuzzing-based Hypertesting

A simple way to generate hypertest inputs is based on fuzzing, where values for the input k-tuples of memories are randomly generated. We expect this strategy to be sufficient to test simple programs, but it may exhibit low detection performance when program complexity increases and, hence, hypercoverage goals are harder to cover. We call such approach *Hyper-Fuzzing*, described in Algorithm 3. The procedure `InputsWithConstraints` generates a random initial set of inputs, consisting of k memories that satisfy the input predicate $\mathbb{P}_i$. Given one (e.g., randomly generated) input memory $\mathbb{m}_1$, the remaining $k-1$ ones needed to create a hypertest input can be obtained by running an SMT solver that solves $\mathbb{P}_i(\mathbb{m}_1, \ldots, \mathbb{m}_k)$, with $\mathbb{m}_2, \ldots, \mathbb{m}_k$ free variables.

The procedure `RunInputs` executes the generated hypertest inputs. It updates the covered hypercoverage goals and it adds the corresponding covering hypertest inputs to the current "archive" of successful inputs (that will be part of the final hypertest suite).

3.2 Hypertesting as an Optimization Problem

As there is no analytical solution to the problem of finding the hypertest inputs that satisfy all hypercoverage goals, we may resort to meta-heuristic search-based algorithm [26], by restating hypertest input generation as an optimization problem. We will first consider its single-objective and then its multi-objective formulation.

Definition 8 (Single-Objective Optimization). *Given a set G of hypercoverage goals, find a set T of hypertest inputs that minimizes the* fitness function *f_G:*

$$\min f_G(T) \triangleq \textstyle\sum_{g \in G} \mathsf{NMRD}(g, T) \quad \textit{where}$$
$$\mathsf{NMRD}(g, T) \triangleq \min\{\mathsf{MRD}(g,t) \mid t \in T\} \,/\, \min\{\mathsf{MRD}(g,t) \mid t \in T\} + 1$$

The fitness function in Definition 8 considers all goals at the same time, aggregating all corresponding distance metrics, yielding a single-objective minimization task. Following [32], we rewrite Definition 8 as a many-objective optimization problem.

Definition 9 (Many-Objective Optimization). *Given a set G of hypercoverage goals, find a set of* non-dominated *hypertest inputs t that minimize the* fitness vector *$\vec{f_G} \triangleq \langle f_g \rangle_{g \in G}$:*

$$\min \vec{f_G} \triangleq \langle \min f_g(t) \triangleq \mathsf{NMRD}(g,t) \rangle_{g \in G} \quad \textit{where}$$
$$\mathsf{NMRD}(g, t) \triangleq \mathsf{MRD}(g,t) \,/\, \mathsf{MRD}(g,t) + 1$$

```
procedure HyperTesting(P, ℙ_i, ℙ_o, k)
1   cfg ← ComputeCFG(P)
2   goals ← ValueInsensitiveCriterion(cfg, ℙ_i, ℙ_o, k)
3   (covered, hypertest) ← HyperTester(P, ℙ_i, k, goals)
4   hypercoverage ← |covered|/|goals|
5   violations ← OracleCheck(ℙ_o, k, hypertest)
6   if violations ≠ ∅ then
7     return (UNSAFE, violations)
    else
      if hypercoverage ≥ THRESHOLD then
8       return (LIKELY_SAFE, hypercoverage)
      else
9       return GIVE_UP
      end
    end
```

Algorithm 2: Hypertesting procedure.

```
procedure HyperFuzz(P, ℙ_i, k, goals)
1   inputs ← InputsWithConstraints(ℙ_i, k)
2   (covered, hypertest) ← RunInputs(P, k, goals, ∅, inputs, ∅)
3   while TestingBudgetNotExpired() do
4     inputs ← InputsWithConstraints(ℙ_i, k)
5     (covered, hypertest) ← RunInputs(P, k, goals, covered, inputs, hypertest)
    end
6   return (covered, hypertest)
```

Algorithm 3: Fuzzing-based hyperproperty testing.

In many-objective optimization, candidate solutions are evaluated in terms of *Pareto dominance* [13], that we can restate in the context of hypertesting as follows.

Definition 10 (Dominance). *A hypertest input t* dominates *another hypertest input t', w.r.t. the fitness vector $\langle f_g \rangle_{g \in G}$, if and only if both the following hold:*

- *$f_g(t) \leq f_g(t')$, for all $g \in G$; and*
- *$f_g(t) < f_g(t')$, for some $g \in G$.*

We write $t \prec_G t'$ to indicate that t dominates t', when the set G of hypercoverage goals is considered.

Among all possible hypertest inputs, the *(Pareto) optimal* ones are those non-dominated by any other possible hypertest input.

Definition 11 (Preference). *Given a hypercoverage goal $g = (\mathrm{x}, \underline{\ell_1}, \ldots, \underline{\ell_k}) \in G$, a hypertest input $t = (\mathbb{m}_1, \ldots, \mathbb{m}_k)$ is* preferred *over another hypertest input $t' = (\mathbb{m}'_1, \ldots, \mathbb{m}'_k)$, w.r.t. the fitness vector $\langle f_{\hat{g}} \rangle_{\hat{g} \in G}$, if and only if one of the following holds:*

- *$f_g(t) < f_g(t')$; or*
- *$f_g(t) = f_g(t') \wedge \min\{\mathsf{SRD}(\underline{\ell_i}, \mathbb{m}_i)\}_{i=1}^{k} < \min\{\mathsf{SRD}(\underline{\ell_i}, \mathbb{m}'_i)\}_{i=1}^{k}$.*

We write $t \preccurlyeq_{\hat{g}} t'$ to indicate that t is preferred over t', when the hypercoverage goal g is considered.

The preference criterion states that one hypertest input is preferred for a goal if it has lower multi-run distance than the other. When two hypertest inputs have the same multi-run distance, we prefer those having the minimum single-run distance. The rationale is that such hypertest input is closer to partially cover a hypercoverage goal (i.e., to cover one of the program point in the goal).

Among all inputs, the *best hypertest input* for a hypercoverage goal is the one preferred over all others for such target. The preference criterion is used for selecting the best non-dominated testcases.

Note that we do not select the best hypertest input covering a goal as the one with least complexity. In fact, the structural complexity of our hypertest inputs (k-tuples) is always the same, as we do not generate sequences of method invocations, just single k-tuples of input memories. Hence, successful hypertest inputs are all considered at the same complexity level.

3.3 Evolutionary-based Hypertesting

Solving the optimization problem presented in Definition 9 results in a clever inspection of the program to check, in order to find potential hyperproperty violations. The more goals we cover and the more counterexamples for the hyperproperty (if any) we expect to find. It has been empirically showed [32] that multi-objective optimization outperforms the single-objective one for test generation. Hence, to solve the problem in Definition 9 we adopt the state-of-the-art many-objective search-based algorithm MOSA [32], with some modifications introduced to adapt it to the hyperproperty setting. These modifications yield what we call *Evolutionary Hypertesting*, described in Algorithm 4.

The algorithm follows the general pattern introduced by MOSA, and the components at lines **6** (preference sorting), **10** (crowding distance assignment) and **13** (final sorting) can be easily derived from the dominance (Definition 10) and preference (Definition 11) relations introduced in Subsection 3.2. The most important modifications w.r.t. standard MOSA are the red-highlighted procedures of Algorithm 4. `InputsWithConstraints` and `RunInputs` are the same procedure described in Subsection 3.1, while `GenerateOffspring` is in charge of generating new individuals (i.e., hypertest inputs), hopefully better ones, to be added to the current population.

In `GenerateOffspring`, we first select two groups of optimal hypertest inputs from the population (*selection phase*), by using the dominance (Definition 10) and preference (Definition 11) relations. Then, we apply crossover and mutation operators specifically designed for hyperproperties.

In the *crossover phase*, pairs of individuals, taken from the two selected groups, are swapped by the *Pair-wise Memory Crossover*, that exchanges the values of a randomly chosen variable between all memories in the two hypertest inputs.

Definition 12 (Pair-wise Memory Crossover). *The* Pair-wise Memory Crossover *operator C for the hypertest input pair $(t, \hat{t})$, with $t = (\mathbb{m}_1, \ldots, \mathbb{m}_k)$ and $\hat{t} = (\hat{\mathbb{m}}_1, \ldots, \hat{\mathbb{m}}_k)$, is:*

$$C(t, \hat{t}) \triangleq \begin{cases} (t', \hat{t}') & \text{if } (t', \hat{t}') = \mathsf{swap}(t, \mathtt{x}, \hat{t}) \wedge \mathbb{P}_i(t') \wedge \mathbb{P}_i(\hat{t}') \\ (t, \hat{t}) & \text{otherwise} \end{cases}$$

for a variable $\mathtt{x} \in \mathsf{vars}(P)$ randomly selected.

Here, the *swap* of $\mathtt{x}$ between the hypertest inputs $t = (\mathbb{m}_1, \ldots, \mathbb{m}_k)$ and $\hat{t} = (\hat{\mathbb{m}}_1, \ldots, \hat{\mathbb{m}}_k)$ is defined as $\mathsf{swap}(t, \mathtt{x}, \hat{t}) \triangleq (t', \hat{t}')$, where:

$$t' = (\mathbb{m}_1[{}^{\hat{\mathbb{m}}_1(\mathtt{x})}/_{\mathtt{x}}], \ldots, \mathbb{m}_k[{}^{\hat{\mathbb{m}}_k(\mathtt{x})}/_{\mathtt{x}}])$$
$$\hat{t}' = (\hat{\mathbb{m}}_1[{}^{\mathbb{m}_1(\mathtt{x})}/_{\mathtt{x}}], \ldots, \hat{\mathbb{m}}_k[{}^{\mathbb{m}_k(\mathtt{x})}/_{\mathtt{x}}])$$

Once the selected individuals have been scrambled, we perform the *mutation phase*, by applying a random value mutation, with probability α. This perturbation implements the *Single Memory Mutation*, which randomly selects a variable and assigns it with a new value, randomly chosen from the variable type.

```
procedure HyperEvo(P, ℙ_i, k, goals)
1   population ← InputsWithConstraints(ℙ_i, k)
2   (covered, hypertest) ← RunInputs(P, k, goals, ∅, population, ∅)
3   while TestingBudgetNotExpired() do
4     offspring ← GenerateOffspring(ℙ_i, k, population)
5     (covered, hypertest) ← RunInputs(P, k, goals, covered, offspring, hypertest)
6     fronts ← PreferenceSorting(goals, covered, population ∪ offspring)
7     newPopulation ← ∅
8     while |newPopulation| + |fronts_rank| ≤ size do
9       currentFront ← CrowdingDistanceAssignment(fronts_rank)
10      newPopulation ← newPopulation ∪ currentFront
11      rank ← rank + 1
      end
12    lastFront ← CrowdingDistanceDescendingSort(fronts_rank)
13    lastIndividuals ← lastFront[1 : (size − |newPopulation|)]
14    population ← newPopulation ∪ lastIndividuals
    end
15  return (covered, hypertest)
```

Algorithm 4: Evolutionary-based hyperproperty testing.

Definition 13 (Single Memory Mutation). *The* Single Memory Mutation *operator $\mathcal{M}$ for the hypertest input $t = (\mathbb{m}_1, \ldots, \mathbb{m}_k)$ is:*

$$\mathcal{M}(t) \triangleq \begin{cases} t' & \text{if } t' = \mathcal{M}(t, j) \wedge \mathbb{P}_i(t') \text{ for a random } j \in [1, k] \\ t & \text{otherwise} \end{cases}$$

where $\mathcal{M}(t, j) \triangleq (\mathbb{m}_1, \ldots, \mathbb{m}_j[{}^{v}/_{\mathtt{x}}], \ldots, \mathbb{m}_k)$, for a random variable $\mathtt{x} \in \mathsf{vars}(\mathbb{m}_j)$ and a value $v \in \mathsf{typeOf}(\mathtt{x})$ randomly generated.

Finally, since hypertest inputs must satisfy the input predicate $\mathbb{P}_i$, the resulting individuals violating $\mathbb{P}_i$ are discarded. An alternative to discarding the individuals that violate $\mathbb{P}_i$ (not yet implemented in our tool) could be to repair them, e.g., by applying an SMT solver to $\mathbb{P}_i$ after replacing some concrete values with free variables.

3.4 Implementation

We have developed two tools that implement the proposed hypertesting procedure (Algorithm 2) for Java, HyperFuzz and HyperEvo, considering one specific hyperproperty, Non-Interference [12]. In particular, HyperFuzz adopts the Hyper-Fuzzing approach of Subsection 3.1, while HyperEvo adopts the Evolutionary Hypertesting approach of Subsection 3.3. Both tools can execute a Java program under two execution scenarios, whose inputs satisfy the input predicate for Non-Interference, i.e., in the two executions all public input variables have the same values, while confidential input variables differ on at least one value. Both can also check the output predicate for Non-Interference, i.e., whether any public output variable has a different value in the two executions, which indicates some information leakage from confidential to public variables. The difference between HyperFuzz and HyperEvo is that the latter uses the hypercoverage-based fitness function described in Section 3.2 as guidance, while the former has no guidance (i.e., it generates random hypertest inputs that satisfy the input predicate).

3.5 Discussion

A potential weak point of our approach consists in the fact that the proposed coverage criterion is a necessary but not sufficient condition to reveal hyperproperty violations, hence our tool may yield false negatives. This is somewhat expected, being our approach

dynamic in nature. Nevertheless, the empirical evaluation we conducted in Section 4 indicates that the approach is indeed effective in spotting hyperproperty violations (at least for Non-Interference).

In addition, our framework targets k-hyperproperties and it may not generalize to other hyperproperties. We started with such subset of hyperproperties since Non-Interference, that is the prominent hyperproperty example, belongs to it (and other important requirements, such as *data races*, are k-hyperproperties). We plan to extend our framework to other kind of hyperproperties as a future work.

4 EMPIRICAL EVALUATION

To empirically validate the value-insensitive hypercoverage criterion and the proposed hypertesting approach, we considered a specific hyperproperty, Non-Interference [12] (in short, there should be no information flow from confidential to public variables), and we have instantiated our framework to test it. We considered Non-Interference among other possibilities because of its relevance and importance for software privacy and security. In our empirical study, we address the following three research questions.

RQ₁ (Correlation): *Is there a relation between high value-insensitive hypercoverage and the detection of Non-Interference violations?*

RQ₂ (Coverage): *Is the proposed approach for hypertest input generation able to achieve high hypercoverage?*

RQ₃ (Effectiveness): *Is the proposed hypertesting technique effective at exposing Non-Interference violations? How does it compare to state-of-the-art dynamic taint analysis?*

With these research questions we gradually validate the hypotheses behind the proposed hypertesting approach: first we check if hypercoverage correlates with the exposure of hyperproperty violations (RQ_1), by adopting a standard correlation metric (Point-Biserial); then, if the proposed hypertest input generators can achieve high coverage (RQ_2), by measuring the amount of hypercoverage goals covered; and, finally, if the hypertest inputs generated under the guidance of hypercoverage can effectively expose hyperproperty violations (RQ_3), by computing standard information retrieval metrics (recall and accuracy). Since there are no dynamic verification approaches specifically targeting Non-Interference providing a tool (see Section 5 for a qualitative comparison with the related work), we compare our approach with a dynamic taint analysis, that is the most similar dynamic technique allowing to track information flows (and, hence, serve as baseline for our tools). For such comparison, we have chosen the state-of-the-art tool Phosphor [4]. Technically, the latter is not purely dynamic, since it leverages static program analysis (e.g., to track control-flow relationships, as described by Hough and Bell [19]).

4.1 Experiment Setting

Program Datasets. In our empirical evaluation, we used the Java classes provided by IFSpec [17], a collection of Java applications that are by design vulnerable or non-vulnerable to Non-Interference. In IFSpec, variables are already tagged with security levels, either public or confidential, by using RIFL [3] specifications. IFSpec is intended to be a benchmark to stress the capabilities of static analyzers that target Non-Interference vulnerabilities. For this reason, the programs in IFSpec make use of a large portion of the syntactic structures provided by Java. Since our implementation does not support yet some of them (e.g., exceptions), we selected the samples in IFSpec that can be managed by our tool instrumentation, resulting in 34 vulnerable and non-vulnerable Java classes (*FullDataset*). To answer the first research question only the vulnerable programs are needed, which amounts to 14 samples (*UnsecureOnlyDataset*).

Experimental Procedure. To answer the previously mentioned research questions, we adopted the following methodology.

(RQ_1). We specifically developed a tool that randomly generates a pool of `POOL_SIZE` = 1000 hypertest inputs for each program of *UnsecureOnlyDataset*. Then, the tool randomly assigns the elements of such pool to groups of size `SAMPLING` = 100. For each group, the tool considers its element in random order, measuring the incremental hypercoverage reached and associating to each level whether a Non-Interference violation was exposed or not. Such hypercoverage level and violation flag pairs have been finally used to compute the Point-biserial correlation.

(RQ_2). Both HyperFuzz and HyperEvo take as input the source code of the Java class under test and a configuration file containing the security level tags for the class and method variables (that we manually retrieved from the RIFL specification present in IFSpec). Then, the tools instrument and compile on-the-fly the input class and perform the hypertesting session. For each program of *FullDataset*, we run HyperFuzz and HyperEvo with the same testing budget of `MAX_CALLS` = 2000 invocations of the method under test. After completion, we collected the reached level of hypercoverage for both tools. Due to non-deterministic components present in the hypertest input generation, each tool (for each program) has been run 5 times, measuring then the average hypercoverage.

(RQ_3). Phosphor requires a manual modification of the program source code, in order to insert the information needed to perform instrumentation. In particular sources (confidential variables in our case) and sinks (public variables in our case) must be wrapped into specific calls to Phosphor's APIs (again, security level tags have been manually retrieved from the RIFL specifications present in IFSpec). For each program of *FullDataset*, we run HyperFuzz, HyperEvo and Phosphor (the first two with a testing budget of `MAX_CALLS` = 2000 invocation of the method under test, while the third does not require any analysis budget to be set). After completion, we retrieved the testing/analysis results for all tools. HyperFuzz and HyperEvo output directly whether Non-Interference violations have been found or not, while Phosphor outputs the possible taint tags of each sinks. We considered a Non-Interference violation for Phosphor as the fact that a public variable is tainted by a label corresponding to a confidential variable (indicating a dependence between the latter and the former). Due to non-deterministic components present in all approaches, each tool (for each program) has been run 5 times.

Collected Metrics. To answer RQ_1 we compute the correlation between the number of value-insensitive hypercoverage goals covered and the detection of a Non-Interference vulnerability. Since in our dataset each sample contains only one vulnerability, the outcome of the testing is binary (violation found or not). Hence, we apply the Point-biserial correlation, a standard correlation coefficient (denoted as R) to be used when one variable is dichotomous.

To answer RQ_2 we compute the coverage level reached by our hypertesting approach, for both HyperFuzz and HyperEvo versions.

To answer RQ_3 we compare the Non-Interference violations (unsafe programs) found during the hypertesting sessions with the ground truth provided by IFSpec. In particular, we adopt the following standard information retrieval metrics.

True Positives (TP) That is, the number of unsafe programs that are correctly detected as unsafe.

False Positives (FP) That is, the number of programs reported as unsafe that correspond to safe ones (i.e., false alarms).

False Negatives (FN) That is, the number of unreported unsafe programs (i.e., missed violations).

True Negatives (TN) That is, the number of safe programs that are correctly detected as safe.

While, by construction, HyperFuzz and HyperEvo, report only hypertests that provably produce a Non-Interference violation, Phosphor might instead report a public variable as incorrectly tainted. This may happen for two reasons: because Phosphor approximates a hyperproperty (Non-Interference) with a trace property (taint propagation); or, because of *dynamic overtainting*, i.e., because it conservatively propagates the taint tag when the information flow is unknown (e.g., when information flows into native code or into black-box library components that cannot be instrumented). Hence, only Phosphor could potentially report false positives.

Since our technique has FP = 0 by construction, to evaluate its accuracy the most interesting metrics are TP and FN, which can be combined into the **True Positives Rate (TPR)** (aka recall). We also compute a single accuracy metric **Accuracy (ACC)**, that aggregates all correct predictions against all performed predictions.

$$\mathrm{TPR} \triangleq \mathrm{TP} \,/\, \mathrm{FN+TP} \qquad \mathrm{ACC} \triangleq \mathrm{TP+TN} \,/\, \mathrm{FN+FP+TP+TN}$$

False positives FP are measured for Phosphor only, as they are by construction zero for HyperFuzz and HyperEvo. In our empirical evaluation, we consider a false (resp. true) negative for HyperFuzz and HyperEvo when they output `LIKELY_SAFE` or `GIVE_UP` for an unsecure (resp. secure) program.

4.2 Experimental Results

Table 1 shows the Point-biserial correlation R between the level of hypercoverage achieved by randomly generated hypertest inputs and the corresponding boolean variable stating whether a Non-Interference vulnerability was exposed or not by the hypertest. The p-values indicate that all correlations are significantly different from 0. Actually, most of them indicate strong correlation, with a value greater than 0.7 (green-highlighted in Table 1). Based on these results, we can formulate the following answer to RQ_1.

> ***RQ_1 (Correlation)*** *The value-insensitive hypercoverage criterion helps in discovering Non-Interference violations, since there is an overall positive and significant correlation between the increasing number of value-insensitive hypercoverage goals covered and the likelihood of detecting a Non-Interference violation.*

Table 2 (*Coverage* columns) shows the number of hypercoverage goals identified in each Java program (column *Goals*) followed by the proportion of such goals covered by HyperFuzz and HyperEvo, respectively. In general, the achieved level of coverage is high for both

Table 1: Correlation results

Sample (UnsecureOnlyDataset)	R	p-value
Aliasing-ControlFlow-u	0.3376	0.0154
Aliasing-InterProcedural-u	0.7071	0.0000
Aliasing-Nested-u	0.7071	0.0000
Aliasing-Simple-u	0.6547	0.0000
Arrays-ImplicitLeak-u	0.5787	0.0000
BooleanOperations-u	1.0000	0.0000
Deepalias-u	0.1286	0.0315
Deepcall-u	1.0000	0.0000
DirectAssignment-u	1.0000	0.0000
DirectAssignmentLeak-u	1.0000	0.0000
HighCond.IncrementalLeak-u	0.7071	0.0000
IFLoop-u	0.5222	0.0000
ScenarioPassword-u	0.5078	0.0000
SimpleArraySize-u	0.8660	0.0000

Weak Positive: $0 < R \leq 0.3$

Positive: $0.3 < R \leq 0.7$

Strong Positive: $0.7 < R \leq 1$

HyperFuzz and HyperEvo, which indicates that on this benchmark both proposed generation strategies (fuzzing and search-based) are generally effective. However, there is also evidence that the search-based strategy can be more effective than fuzzing: on *Aliasing-ControlFlow-u* and *Arrays-ImplicitLeak-u* HyperEvo achieves 100% coverage, while HyperFuzz achieves respectively 67% and 62% coverage. We manually investigated the reasons for such a difference and found that the hypercoverage goals missed by fuzzing require a smart selection of the confidential inputs values, because a specific path, yielding when traversing the code guarded by a non-trivial conditional, has to be taken to reach them.

There are also two instances in which full coverage is not reached, neither by HyperEvo nor by HyperFuzz, that are *ScenarioPassword-s* and *ScenarioPassword-u*. We manually investigated the hypercoverage goals missed by both implementations of our approach and found that they are both infeasible goals (Definition 5), hence associated with paths that cannot be covered by any pair of inputs satisfying the input predicate. Based on these results, we can formulate the following answer to RQ_2.

> ***RQ_2 (Coverage)*** *The proposed hypertesting approach is very effective in covering value-insensitive hypercoverage goals, since it obtains full-coverage in the majority of the considered case studies and, overall, the coverage reached is never less than 43%.*

Table 2 (*Violations* columns) shows the ground truth classification of each Java program, which can be *secure* (✓) or *unsecure* (✗). The outcome of each tool being compared is reported in the following columns. In particular, column Phosphor reports *secure* (✓) (resp. *unsecure* (✗)) when the taint tag associated with confidential variables propagates to a public variable in a program execution, while columns HyperFuzz and HyperEvo report *secure* (✓) (resp. *unsecure* (✗)) when an automatically generated hypertest input provides a counterexample violating Non-Interference, i.e., the tool generated a pair of executions differing only on the value of some confidential input variables, eventually affecting the value of some public output variables, which differ between the two executions.

We can notice that in many cases there is agreement with the ground truth across the three tools: they all find the vulnerability, if present, or report no alarm if the code is secure. Disagreements with ground truth are indicated with a red background. There are

Table 2: Hypercoverage and Accuracy results

Sample (FullDataset)	*Coverage (RQ_2)*			*Violations (RQ_3)*			
	Goals	HyperFuzz	HyperEvo	*Ground Truth*	Phosphor	HyperFuzz	HyperEvo
Aliasing-ControlFlow-s	6	1.00	1.00	✓	✓	✓	✓
Aliasing-ControlFlow-u	6	0.67	1.00	✗	✓	✓	✗
Aliasing-InterProcedural-s	4	1	1	✓	✓	✓	✓
Aliasing-InterProcedural-u	4	1	1	✗	✗	✗	✗
Aliasing-Nested-s	5	1	1	✓	✓	✓	✓
Aliasing-Nested-u	4	1	1	✗	✗	✗	✗
Aliasing-Simple-s	3	1	1	✓	✓	✓	✓
Aliasing-Simple-u	5	1	1	✗	✗	✗	✗
Aliasing-StrongUpdate-s	7	1	1	✓	✓	✓	✓
ArrayIndexSensitivity-s	3	1	1	✓	✓	✓	✓
ArraySizeStrongUpdate-s	4	1	1	✓	✓	✓	✓
Arrays-ImplicitLeak-s	8	1	1	✓	✓	✓	✓
Arrays-ImplicitLeak-u	8	0.62	0.97	✗	✓	✓	✗
BooleanOperations-s	1	1	1	✓	✓	✓	✓
BooleanOperations-u	1	1	1	✗	✓	✗	✗
CallContext-s	3	1	1	✓	✓	✓	✓
Deepalias-s	27	1	1	✓	✓	✓	✓
Deepalias-u	27	1	1	✗	✗	✗	✗
Deepcall-s	1	1	1	✓	✓	✓	✓
Deepcall-u	1	1	1	✗	✗	✗	✗
DirectAssignment-s	1	1	1	✓	✓	✓	✓
DirectAssignment-u	1	1	1	✗	✗	✗	✗
DirectAssignmentLeak-u	1	1	1	✗	✗	✗	✗
HighCond.Incr.Leak-s	1	1	1	✓	✓	✓	✓
HighCond.IIncr.Leak-u	3	1	1	✗	✓	✗	✗
IFLoop-s	16	1	1	✓	✓	✓	✓
IFLoop-u	9	1	1	✗	✗	✗	✗
IFMethodContractA-s	7	1	1	✓	✓	✓	✓
IFMethodContractB-s	7	1	0.97	✓	✓	✓	✓
LostInCast-s	4	1	1	✓	✗	✓	✓
ScenarioPassword-s	7	0.43	0.43	✓	✓	✓	✓
ScenarioPassword-u	12	0.50	0.50	✗	✓	✗	✗
SimpleArraySize-u	2	1	1	✗	✗	✗	✗
Simp.Eras.ByCond.Checks-s	11	1	1	✓	✓	✓	✓
				TPR	0.64	0.86	1.00
				ACC	0.82	0.94	1.00

two instances in which both Phosphor and HyperFuzz miss the vulnerability, while HyperEvo can detect it: *Aliasing-ControlFlow-u* and *Arrays-ImplicitLeak-u*. Not surprisingly, these are the same two cases where HyperEvo achieved higher hypercoverage than HyperFuzz, which shows the usefulness of hypercoverage as adequacy criterion for hyperproperty testing: by achieving 100% hypercoverage, HyperEvo can also expose the vulnerabilities in these two Java programs, while HyperFuzz misses some hypercoverage goals and correspondingly misses also the vulnerability present in these two programs. It is interesting that Phosphor misses these two vulnerabilities as well. In fact, Phosphor does not rely on hypercoverage. Actually, the taint propagation path that would lead Phosphor to expose the vulnerability was manually found to be also involved in the hypercoverage goals missed by HyperFuzz, thus confirming that a search-based strategy might be needed to look for inputs that exercise specific, vulnerable paths. In three more cases Phosphor missed the vulnerability, while HyperFuzz and HyperEvo are able to detected it: *BooleanOperations-u*, *HighCond.IncrementalLeak-u* and *ScenarioPassword-u*. By manually investigating these cases, we found that taint tags should have been propagated along a path that involves an *implicit* information flow (e.g., inside a loop). Implicit flows are hard to catch by using (not-hyper) dynamic techniques, that use syntactic dependencies to approximate Non-Interference.

Overall, HyperEvo achieves 100% TPR and ACC, HyperFuzz 86% TPR (94% ACC) and Phosphor 64% TPR (82% ACC). By construction, all vulnerabilities reported by HyperFuzz and HyperEvo cannot be false alarms, as the executions exposing the vulnerability are explicitly run and checked to be true positives during the test generation process. On the contrary, dynamic taint analysis is potentially subject to false alarms, in case of over-tainting. Indeed, Phosphor erroneously detects the program *LostInCast-s* as vulnerable. By manually investigating this case, we found that the information flow from a confidential variable to a public one is nullified, at some point during program execution, by a *cast* operation (that drops the four most significant bytes of the confidential variable). Such kind of alarms are quite hard to rule out without comparing two executions of the program and, indeed, Phosphor conservatively marks such syntactic dependency as a (potential) violation. Based on the results, we can provide the following answer to RQ_3.

> ***RQ_3 (Effectiveness)*** *The proposed hypertesting approach is very accurate in detecting Non-Interference vulnerabilities, since it outperforms state-of-the-art dynamic taint analysis. In particular,* HyperFuzz *and* HyperEvo *reach an accuracy of 94% and 100%, respectively, while* Phosphor *reaches an accuracy of 82% only.*

4.3 Threats to Validity

Internal Validity. Internal validity threats are due to the metrics chosen to answer the research questions. We adopted standard metrics from statistics (correlation), structural testing (coverage) and information retrieval (true positive rate, accuracy) that are directly related to the respective research questions. However, different metrics might provide different insights and view points.

External Validity. External validity threats are associated with the generalizability of our findings beyond the considered benchmark. We do not claim any form of general validity of our results beyond the benchmark and we believe that future replications and extensions of the empirical study are needed to corroborate our findings. We chose a subset of the standard benchmark IFSpec, such that our tools could be applied to it, resulting in 34 Java programs.

5 RELATED WORK

Among all works in V&V, only some static approaches *systematically* deal with hyperproperties, in particular using abstract interpretation [2, 23] or model-checking [16, 20]. The latter define a *hyperlogic*, i.e., a temporal logic quantifying over sets of executions. Unfortunately, only a small fragment of this logic is decidable, hence statically verifiable. The drawback of static approaches to hyperproperty verification is their imprecision: hyperproperties are often quite complex, hence resulting in a very coarse analysis. Indeed, a dynamic approach would be potentially more effective but, to the best of our knowledge, there are only a few dynamic methods designed to verify hyperproperties [18, 27–31].

Muduli et al. [29] use fuzzing to generate test cases for generic hyperproperties in the context of Systems-on-Chip (SoC). The approach randomly generates pairs of inputs and checks pairs of executions, but test generation is unguided (there is no target hypercoverage adequacy criteria, as in our approach) and the proposed technique is designed for a very narrow application context (SoC), which makes it difficult to compare with our approach.

Fuzzing-based approaches, such as DifFuzz [30], ct-fuzz [18] and QFuzz [31], test programs against *side-channel leaks*, that are hyperproperties (e.g., timing guarantee). Nevertheless, Non-Interference violations and side-channel leaks are not in general comparable. In such fuzzers, test generation, which exploits either multi-executions [7, 30, 31] or self-composition [18], is essentially random, and not guided by the hyperproperty to test as in our approach. Since test generation in such works is random, hence similar to that implemented in HyperFuzz, we believe that our empirical results support already an indirect comparison with these approaches.

Concerning *information-flows*, the closest work is HyperGI [28], a technique that uses multiple program executions to measure information leaks and to repair them by using genetic improvement. By resorting to entropy-based measures, HyperGI checks *Quantitative Non-Interference* [9, 35]. Nevertheless, their focus is on program repair rather than test input generation – our paper's focus. Indeed, in HyperGI input generation is based on a binary search, that iteratively halves the input space and selects some public inputs from each half. Then, confidential inputs are altered, in order to spot changes in the output. Since they hold several similarities, we could have compared HyperGI with HyperFuzz and HyperEvo, but, unfortunately, the tool is not available. HyperGI's follow-up work [27] proposes LeakReducer, that improves the repair phase of HyperGI by adopting a multi-objective approach, keeping unchanged the test input generation phase. Again, we could have compared LeakReducer with HyperFuzz and HyperEvo, but the tool is not available. Finally, some other V&V works verify Non-Interference by using abstract interpretation [24] or hybrid monitors [22].

Metamorphic Testing. As in our approach, *Metamorphic Testing (MT)* [8] also exploits multiple program executions. In MT some necessary properties of the program are identified, taking the form of *metamorphic relations (MR)* among multiple inputs and their expected outputs. Such relations are used to transform existing (*source*) test cases into new (*follow-up*) ones, which by construction satisfy the input part of the MR. A bug is found when source and follow-up test cases satisfy the input but not the output part of a MR. Indeed, MT was proposed as a method to alleviate the oracle problem when testing programs whose expected behaviour is difficult or impossible to anticipate (e.g., machine learning techniques). Even when a thorough oracle cannot be defined, if the actual outputs of source and follow-up tests violate a certain MR, we can say that the program under test is faulty w.r.t. the program property associated with that relation.

In this respect, a metamorphic relation can be seen as a particular k-hyperproperty. However, MT does not provide any guidance on how to verify/refute such relation, in contrast to our framework that derives a hypercoverage adequacy criterion from the hyperproperty, in order to craft specific inputs that may refute it. We believe that our framework may help in improving MT approaches, by providing them with a guiding adequacy criterion for MR violation. Our framework represents a step toward the systematic testing of k-hyperproperties, which include metamorphic relations.

Mutation Testing. Mutation testing [1] can be formulated as a hyperproperty problem, where multi-executions are given by the mutated and the non-mutated versions of the program and the predicate to check is equality. Indeed, Fellner et al. [14] exploit such correspondence to reuse hyperproperty formal verification machinery (e.g., model-checking) to perform mutation testing. So, differently from us, their goal is not to check a hyperproperty but, rather, to improve mutation testing. As mutation testing can be encoded into a hyperproperty, we may use our approach to craft inputs suitable for improving mutation testing as well.

6 CONCLUSION

We have proposed a novel testing framework for hyperproperty testing, consisting of an adequacy criterion, a structural search metaheuristic and a test generation approach. The adequacy criterion, called hypercoverage, was designed to force the exploration of the different variable value assignments, possibly involved in a hyperproperty violation. The test generation approach has two instances, HyperFuzz and HyperEvo, respectively based on fuzzing and search algorithms. The latter takes advantage of the proposed distance metaheuristic to lead test generation to the satisfaction of a hypercoverage goal, which allowed us to formulate hyperproperty testing as an optimization problem in the hypertest input space, solved by the HyperEvo multi-objective search algorithm.

Experimental results confirmed the validity of our framework, at least for the hyperproperty considered in the evaluation (i.e., Non-Interference), by showing that inputs achieving high hypercoverage have a higher chance of exposing hyperproperty violations, and that both tools HyperFuzz and HyperEvo achieve high hypercoverage and correspondingly detect a high number of vulnerabilities in the considered benchmark. They both outperformed the state-of-the-art dynamic taint analysis tool Phosphor. Between them, HyperEvo showed marginal advantages on Java programs that require specific input combinations both to reach the target hypercoverage goals and to expose the vulnerabilities contained in these programs.

Even if the empirical evaluation has been conducted on a specific hyperproperty (i.e., Non-Interference), we believe that the proposed framework is applicable to any k-bounded hyperproperty [23]. In future work, we want to extend the applicability of HyperFuzz and HyperEvo to other hyperproperties, beyond Non-Interference, and we want to test them on additional, more complex, programs.

DATA AVAILABILITY

All material necessary to replicate our experiments is available at: https://github.com/ICSE2024-2ndCycle-Paper123/ReplicationPackage

ACKNOWLEDGMENTS

This work was partially supported by the Italian Ministry of University and Research, under agreement 40-G-14702-3 for the PON programme for Research and Innovation (Action IV.6); the NextGenerationEU project SMARTITUDE, funded by the Italian Ministry of University and Research under the PRIN 2022 programme (Code: D53D23008400006); the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703); and the iNEST project, funded under the PNRR programme (Missione 4, Componente 2, Investimento 1.5 - D.D. 1058 23/06/2022, ECS00000043). This manuscript reflects only the authors' views and opinions, the sponsoring agencies cannot be held responsible for them and any use which may be made of the information contained therein.

REFERENCES

[1] Allen Troy Acree, Timothy Alan Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick Gerald Sayward. 1979. *Mutation Analysis.* techreport GIT-ICS-79/08. Georgia Institute of Technology, Atlanta, Georgia.

[2] Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. 2017. Hypercollecting semantics and its application to static analysis of information flow. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20,2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 874–887. https://doi.org/10.1145/3009837.3009889

[3] Thomas Bauereiß, Simon Greiner, Mihai Herda, Michael Kirsten, Ximeng Li, Heiko Mantel, Martin Mohr, Matthias Perner, David Schneider, and Markus Tasch. 2017. *RIFL 1.1: A Common Specification Language for Information-Flow Requirements.* Technical Report. TU Darmstadt. 46.12.03; LK 01.

[4] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

[5] Raven Beutner and Bernd Finkbeiner. 2023. HyperATL*: A Logic for Hyperproperties in Multi-Agent Systems. *Log. Methods Comput. Sci.* 19, 2 (2023). https://doi.org/10.46298/lmcs-19(2:13)2023

[6] D. W. Binkley and K. B. Gallagher. 1996. Program Slicing. *Advances in Computers* 43 (1996), 1–50. https://doi.org/10.1016/S0065-2458(08)60641-5

[7] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1011–1023. https://doi.org/10.1145/3377811.3380432

[8] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report. The Hong Kong University of Science and Technology. HKUST-CS98-01.

[9] David Clark, Sebastian Hunt, and Pasquale Malacaria. 2002. Quantitative Analysis of the Leakage of Confidential Data. *Electronic Notes in Theoretical Computer Science* 59, 3 (2002), 238–251. https://doi.org/10.1016/S1571-0661(04)00290-7 QAPL'01, Quantitative Aspects of Programming Laguages (Satellite Event of PLI 2001).

[10] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–284.

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

[12] E. Cohen. 1977. Information Transmission in Computational Systems. *Oper. Syst. Rev.* 11 (1977), 133–139.

[13] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 577–601. https://doi.org/10.1109/TEVC.2013.2281535

[14] Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2019. Mutation Testing with Hyperproperties. In *Software Engineering and Formal Methods*, Peter Csaba Ölveczky and Gwen Salaün (Eds.). Springer International Publishing, Cham, 203–221.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[16] Bernd Finkbeiner. 2021. Model Checking Algorithms for Hyperproperties (Invited Paper). In *Verification, Model Checking, and Abstract Interpretation*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer International Publishing, Cham, 3–16.

[17] Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. 2018. A Uniform Information-Flow Security Benchmark Suite for Source Code and Bytecode. In *Secure IT Systems - 23rd Nordic Conference, NordSec 2018, Oslo, Norway, November 28-30, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11252)*, Nils Gruschka (Ed.). Springer, 437–453. https://doi.org/10.1007/978-3-030-03638-6_27

[18] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 466–471. https://doi.org/10.1109/ICST46399.2020.00063

[19] Katherine Hough and Jonathan Bell. 2021. A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 26 (dec 2021), 43 pages. https://doi.org/10.1145/3485464

[20] T. Hsu, C. Sánchez, and B. Bonakdarpour. 2021. Bounded Model Checking for Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen (Eds.). Springer International Publishing, Cham, 94–112.

[21] Johannes Kinder. 2015. Hypertesting: The Case for Automated Testing of Hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*. 1–8.

[22] G. Le Guernic. 2008. Precise Dynamic Verification of Confidentiality. In *Proceedings of the 5th International Verification Workshop in connection with IJCAR 2008, Sydney, Australia, August 10-11, 2008 (CEUR Workshop Proceedings, Vol. 372)*, B. Beckert and G. Klein (Eds.). CEUR-WS.org.

[23] I. Mastroeni and M. Pasqua. 2018. Verifying Bounded Subset-Closed Hyperproperties. In *Static Analysis*, Andreas Podelski (Ed.). Springer International Publishing, Cham, 263–283.

[24] I. Mastroeni and M. Pasqua. 2019. Statically Analyzing Information Flows: An Abstract Interpretation-Based Hyperanalysis for Non-Interference. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Association for Computing Machinery, 2215–2223.

[25] Isabella Mastroeni and Michele Pasqua. 2022. Verifying Opacity by Abstract Interpretation. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22)*. Association for Computing Machinery, New York, NY, USA, 1817–1826. https://doi.org/10.1145/3477314.3507119

[26] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156. https://doi.org/10.1002/stvr.294

[27] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra Cohen, and Justyna Petke. 2023. Keeping Secrets: Multi-Objective Genetic Improvement for Detecting and Reducing Information Leakage. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 61, 12 pages. https://doi.org/10.1145/3551349.3556947

[28] I. Mesecan, D. Blackwell, D. Clark, M. B. Cohen, and J. Petke. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1358–1362.

[29] S. K. Muduli, G. Takhar, and P. Subramanyan. 2020. Hyperfuzzing for SoC Security Validation. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, 9 pages.

[30] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DifFuzz: Differential Fuzzing for Side-Channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 176–187. https://doi.org/10.1109/ICSE.2019.00034

[31] Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: Quantitative Fuzzing for Side Channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 257–269. https://doi.org/10.1145/3460319.3464817

[32] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435

[33] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 392–404. https://doi.org/10.1109/CSF.2017.13

[34] Shubham Sahai, Pramod Subramanyan, and Rohit Sinha. 2020. Verification of Quantitative Hyperproperties Using Trace Enumeration Relations. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 201–224.

[35] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computational Structures*, Luca de Alfaro (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–302.

[36] Yu Wang, Siddhartha Nalluri, and Miroslav Pajic. 2020. Hyperproperties for Robotics: Planning via HyperLTL. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 8462–8468. https://doi.org/10.1109/ICRA40945.2020.9196874

[37] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136