Specification of Concretization and Symbolization Policies in Symbolic Execution[*]

Robin David, Sébastien Bardin
CEA, LIST, Saclay, France
first.last@cea.fr

Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta
UGA, Verimag, Grenoble, France
first.last@imag.fr

Jean-Yves Marion
Université de Lorraine, CNRS
LORIA, Nancy, France
first.last@loria.fr

ABSTRACT

Symbolic Execution (SE) is a popular and profitable approach to automatic code-based software testing. Concretization and symbolization (C/S) is a crucial part of modern SE tools, since it directly impacts the trade-offs between correctness, completeness and efficiency of the approach. Yet, C/S policies have been barely studied. We intend to remedy to this situation and to establish C/S policies on a firm ground. To this end, we propose a clear separation of concerns between C/S specification on one side, through the new rule-based description language CSml, and the algorithmic core of SE on the other side, revisited to take C/S policies into account. This view is implemented on top of an existing SE tool, demonstrating the feasibility and the benefits of the method. This work paves the way for more flexible SE tools with well-documented and reusable C/S policies, as well as for a systematic study of C/S policies.

CCS Concepts

•**Software and its engineering** → **Formal software verification;** *Software testing and debugging;* Dynamic analysis; Specification languages;

Keywords

automatic test generation; formal methods; symbolic execution; specification language

1. INTRODUCTION

Context. Symbolic Execution (SE) [16] is a popular and fruitful formal approach to automatic (code-based) software testing. Given a path in a program, the key insight of SE is the possibility in many cases to compute a formula (a *path predicate*) such that a solution to this formula is a test input exercising the considered path. Then, exploring all the (bounded) paths of the program allows for intensive testing and efficient bug finding.

Basis were laid in the 70's by King [25], but the technique found a renewal of interest in the mid 2000's when it was mixed with concrete execution [27, 22, 29] and combined with the growing efficiency of SMT solvers. SE has quickly become the most promising technique for code-based automatic test generation, leading to impressive case studies [12, 17, 2, 14] and a promise of industrial adoption at large scale [9, 23]. Its usage for security purposes have also been considered, especially because of its straightforward adaptation to binary-level analysis [23, 4, 28, 3]. SE has successfully been applied in a wide range of security applications, such as vulnerability [1, 24] or malware analysis [10].

Problem. While a purely symbolic approach is worth considering, the strength of modern SE tools is to symbolically evaluate only a (small) trace fragment. *Concretization* uses run-time values in order to under-approximate the path predicate, while *symbolization* over-approximate the path predicate by introducing new logical variables. The former allows to handle in a precise but limited way parts of an execution which are either missing (e.g., system calls) or too costly to reason about (e.g., cryptographic functions), while the latter allows to generalize certain program steps, keeping the reasoning exhaustive but less precise.

Actually, choices of concretization and symbolization (C/S) are a crucial part of modern SE tools, together with path predicate computation and path selection. Yet, while the latter are either well-understood (path predicate) or under active research efforts (path selection), C/S policies have been much less studied. Especially, design choices behind implemented C/S policies are often barely explained, and most SE tools either propose only hard-coded C/S policies, or give full control on a line per line manner in the code [19].

Goal and contribution. We propose to address these problems through a clear separation of concerns between (1) a specification mechanism for C/S policies in SE, and (2) a new SE algorithm parametrized by an arbitrary C/S policy. The main contributions of this paper are the following:

- We formalize what a C/S policy is and *we revisit the standard path predicate computation algorithm taking C/S policy into account* (Section 4.1). This is the first time such a parametric view of the core algorithm behind SE is provided. We clearly show where the C/S policy matters and we discuss correctness issues.
- We propose CSml, a *rule-based specification language for defining C/S policies*, together with its semantics

[*]Work partially funded by ANR, grant 12-INSE-0002.

(Sections 4.2 to 4.4). The language is simple, yet powerful enough to encode standard C/S policies. Again, correctness issues are discussed.

- As a first application, we perform an *extensive literature review*, and we show how to encode existing C/S policies into CSML, highlighting in some cases subtle differences between similar policies (Section 4.5).
- As a second application, these results have been implemented on top of the BINSEC framework [20, 21], yielding *the first SE tool with fully customizable C/S policies through high-level specifications* (Section 5). First experiments demonstrate that the *overhead induced by this genericity is very low* (Section 6.2). We also show an example of an original C/S policy fine-tuned for vulnerability detection (Section 6.3).
- Finally, we present *the first quantitative comparison of C/S policies* (Section 6.1), focused on policies dedicated to the handling of memory operations. We compare five policies on 169 programs. We found that, while new policy PP* performs better on most examples, there is still a high variability of results between the policies depending on the considered example. This is a strong *a posteriori* argument for a generic C/S specification mechanism.

Outcome. This work proposes a clear separation of concerns between the core SE algorithms and C/S specification, paving the way for flexible SE tools with easy to configure C/S policies. Additional benefits include: (1) better documented policies, facilitating their understanding, comparison and reuse; (2) the systematic study of concretization and symbolization (including both analytic and quantitative analysis) in order to better understand their impact and to identify interesting trade-offs; and finally (3) the fine-tuning of dedicated policies tailored to specific programs or needs.

2. BACKGROUND

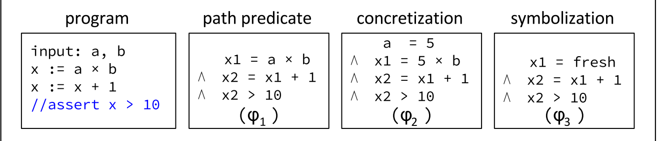
2.1 Notation

Given a program P over a vector V of m input variables taking values in a domain $D \triangleq D_1 \times \ldots \times D_m$, a test datum t for P is a valuation of V, i.e. $t \in D$. The execution of P over t, denoted $P(t)$, is a path (or run) $\sigma \triangleq (l_1, \Sigma_1) \ldots (l_n, \Sigma_n)$, where the l_i denote control-locations (or simply locations) of P and the Σ_i denote the successive internal states of P ($\approx$ valuation of all global and local variables as well as memory-allocated structures) before the execution of each l_i.

2.2 Symbolic Execution in Brief

We recall here a few basic facts about Symbolic Execution (SE) [25] and Dynamic Symbolic Execution (DSE) [22, 27, 29]. Let us consider a program under test P with input variables V over domain D and a path σ of P. The key insight of SE is that it is possible in many cases to compute a *path predicate* ϕ_σ for σ such that for any input valuation $t \in D$, we have: t satisfies ϕ_σ iff $P(t)$ covers σ. Such a path predicate is said to be both *correct* and *complete*, where *correctness* denotes the left-to-right implication (a solution does cover the intended path) and *completeness* denotes the right-to-left implication (any input covering the path is a solution).

A path predicate is intuitively the logical conjunction of all branching conditions and assignments encountered along that path. Figure 1 presents a simple program path (two assignments and a branching condition `x > 10` taken to *true*) together with three possible path predicates. It is straightforward to check that φ_1 is correct and complete (a valuation of `a` and `b` satisfies φ_1 iff their execution satisfies the assertion), while φ_2 is correct but incomplete because of the additional constraint $a = 5$ and φ_3 is complete but incorrect because of the removal of constraint $x_1 = a \times b$.



```
input: a, b
x := a × b
x := x + 1
//assert x > 10
```

```
    x1 = a × b
∧  x2 = x1 + 1
∧  x2 > 10
```

```
    a  = 5
∧  x1 = 5 × b
∧  x2 = x1 + 1
∧  x2 > 10
```

```
    x1 = fresh
∧  x2 = x1 + 1
∧  x2 > 10
```

φ_1 is a correct and complete path predicate, φ_2 is obtained through concretization of `a` (assuming its runtime value is 5) and φ_3 through symbolization of `x` on first line (fresh is a new unconstrained variable)

Figure 1: Path predicate, concretization and symbolization

In practice, path predicates are often under-approximated and only correctness holds, which is fine for testing: SE outputs a set of pairs (t_i, σ_i) such that each t_i is ensured to cover the corresponding σ_i. DSE enhances SE by interleaving concrete and symbolic executions. The dynamically collected information can help the symbolic step, for example by suggesting relevant approximations (cf. Section 2.4).

A high-level view of SE is depicted in Algorithm 1. We assume that the set of paths of P, denoted $Paths(P)$, is finite[1]. The algorithm builds iteratively a set of tests by exploring all the feasible paths.

Algorithm 1: Symbolic Execution algorithm

```
   Input: a program $P$ with finite set of paths $Paths(P)$
   Output: $TS$ a set of pairs $(t, \sigma)$ such that $P(t)$ covers $\sigma$
 1 $TS := \emptyset$;
 2 $S_{paths} := Paths(P)$;
 3 while $S_{paths} \neq \emptyset$ do
 4     choose $\sigma \in S_{paths}$; $S_{paths} := S_{paths} \setminus \{\sigma\}$ ;
 5     compute path predicate $\phi_\sigma$ for $\sigma$ ;
 6     switch $solve(\phi_\sigma)$ do
 7         case $sat(t)$:  $TS := TS \cup \{(t, \sigma)\}$;
 8         case $unsat$:  skip ;
 9     endsw
10 end
11 return $TS$;
```

The three major components of the SE algorithm are the following: (1) path selection strategy (line 4); (2) path predicate computation, with predicates in some theory T; (3) satisfiability checking, using a solver taking a formula $\phi \in T$ and returning either *sat* with a solution t or *unsat*[2]. *We focus in this article on path predicate computation*, which is where C/S policy matters most. Note that the effective solvability of a path predicate may depend a lot on its construction.

From now on, we do not distinguish between SE and DSE.

[1]This assumption is enforced through a bound on paths.
[2]SE tools typically rely on off the shelf SMT solvers.

2.3 Path Predicate Computation

In order to remain both general and concrete, we present path predicate computation on a small core language well-suited to low-level analysis. We choose DBA [21, 6] that has been used in several binary-level analyzers [6, 21, 7, 5]. The core language is presented in Table 1, where Var denotes a set of variables (typically: registers) and Val denotes the set of values, here bitvectors of statically known size. The program is represented as a map from (control) locations to instructions. Operator @ represents both reads at and writes to a distinguished variable Mem (modeling the memory), depending whether they are in a lhs (write) or in an expression (read). All basic bit-level operations are available, including machine arithmetic and bitwise logical operators. The set of instructions includes assignments, static jumps, computed jumps (with an implicit cast from Val to Loc), conditional jumps and a stop operation. We denote by l, v and bv some elements of Loc, Var and Val.

program	::=	ε | stmt program
stmt	::=	$< l, inst >$
inst	::=	lhs := expr
		| goto expr | goto l
		| ite(expr)? ; goto l ; goto l
		| stop
lhs	::=	v | @ $expr$
expr	::=	$expr \ \diamond_b \ expr$ | $\diamond_u \ expr$ | @ $expr$
		| v | bv
$\diamond_u$	::=	$\neg$ | $-$
$\diamond_b$	::=	$+$ | $-$ | $\times_{u,s}$ | $/_{u,s}$ | $\leq_{u,s}$ | $\oplus$ ...

Table 1: DBA instructions

In the sequel $\mathbb{I}nstr$ denotes the set of instructions and $\mathbb{E}xpr$ the set of expressions. The map from locations to instructions is denoted $\Delta : Loc \to \mathbb{I}nstr$. The operational semantics is given in a standard way, each instruction updates a concrete memory state Σ and moves control to the next instruction. Here, Σ is a total function mapping each variable $v \in Var$ to a value $bv \in Val$ (respecting size constraints), and mapping variable Mem to an array from addresses (values of size addr_size) to bytes (values of size 8).

Path predicate computation. We denote by Σ^* the symbolic memory state which maps all variables $v \in Var$ to symbolic values φ (logical terms on logical variables ranging over Val) and the distinguished variable Mem to a logical array from addresses to bytes. The path predicate ϕ is a first-order logic formula over logical variables and logical arrays. At a given point of execution, the internal state of the algorithm is composed of l, Σ^*, ϕ, respectively a location, a symbolic memory state and a (current) path predicate. The algorithm starts from the initial location l_0, *the initial state* Σ_0^* *associating a fresh logical variable to each program variable and a fresh logical array to* Mem, and $\phi_0 \triangleq$ true. It proceeds instruction by instruction along the execution, then the computed path predicate ϕ is returned. Recall that the execution trace being fixed, the successor of each branching instruction is known.

The path predicate computation algorithm over DBA is given in Figure 2, where $\rightsquigarrow$ represents path predicate computation itself while $\vdash_e$ represents the symbolic evaluation of a DBA expression. Recall that φ (resp. ϕ) denotes a symbolic value (resp. a formula). For instance, rule var $\frac{}{\Sigma^*, v \vdash_e \Sigma^*(v)}$ states that the symbolic evaluation of variable v is the symbolic value stored for v in the current symbolic memory state Σ^*, denoted $\Sigma^*(v)$. Rule $l_e - goto\ e$ allows the symbolic evaluation of a dynamic jump branching to location l_e. The rule reads as follows: expression e is symbolically evaluated into the symbolic value φ_e, l_e is converted to a concrete address with $to\_val : Loc \to Val$ and the constraint $to\_val(l_e) = \varphi_e$ is added to the path predicate being built, modeling the fact that the execution flow must lead to l_e. Remaining unexplained symbols are:

- the symbols gathered into $\diamond_u^*$ and $\diamond_b^*$ are the logical counterparts of the unary and binary symbols of concrete expressions, e.g "+" is evaluated to the logical operator `bvadd` of the bitvector theory;
- select/store are the standard logical operators from the theory of arrays, representing reads at and writes to specific array indexes;
- "fresh" designates a new logical variable in the formula.

Property 1. *The path predicate computation algorithm of Figure 2 is correct and complete, i.e. it returns a correct and complete path predicate.*

2.4 Concretization & Symbolisation

In practice, performing a fully symbolic path predicate computation as shown in Figure 2 is not necessarily feasible for various reasons: unavailable parts of the code, presence of an environment, concrete operators outside the scope of the underlying solver, etc. That is why concretization and symbolization were introduced into symbolic execution [27, 22, 19] (cf. Figure 1 for examples):

Concretization uses run-time values in order to *under-approximate* the path predicate, allowing to handle in a precise but limited way parts of an execution which are either missing (e.g., system calls) or too costly to reason about. For instance, concretizing read and write addresses significantly reduces the complexity of the path predicate since the theory of arrays is computationally hard to solve;

Symbolization *over-approximates* the path predicate by introducing a fresh logical variable, allowing to generalize certain program steps, keeping the reasoning exhaustive but less precise. For example, symbolizing `eax` after a system call is a good way to simulate all possible return values of the call;

Propagation computes the path predicate as explained in Section 2.3, without any extra-approximation.

Both concretization and symbolization allow to make SE more robust to real programs, yet they come at the price of losing either completeness (concretization) or correctness (symbolization). The decision upon which a value is concretized or symbolized is in general *hard-coded* inside the path predicate computation, and many alternative choices exist in the literature (cf. Section 7), more or less documented. *Our goal is precisely to design a flexible and clear specification mechanism for such a decision.*

$$\mathbb{E}\mathbf{xpr}: \quad cst\ \frac{}{\Sigma^*, bv \vdash_e bv} \quad var\ \frac{}{\Sigma^*, v \vdash_e \Sigma^*(v)} \quad binop\ \frac{\Sigma^*, e_1 \vdash_e \varphi_1 \quad \Sigma^*, e_2 \vdash_e \varphi_2 \quad \varphi \triangleq \varphi_1 \diamond_b^* \varphi_2}{\Sigma^*, e_1 \diamond_b e_2 \vdash_e \varphi}$$

$$unaryop\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \varphi' \triangleq \diamond_u^* \varphi_e}{\Sigma^*, \diamond_u e \vdash_e \varphi'} \quad @\ \frac{\Sigma^*, e \vdash_e \varphi}{\Sigma^*, @\ e \vdash_e select(\Sigma^*(Mem), \varphi)}$$

$$\mathbb{I}\mathbf{nstr}: \quad goto\ l_1\ \frac{}{l, \Sigma^*, \phi, goto\ l_1 \rightsquigarrow l_1, \Sigma^*, \phi, \Delta(l_1)} \quad l_e - goto\ e\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \phi' \triangleq \phi \wedge to\_val(l_e) = \varphi_e}{l, \Sigma^*, \phi, goto\ e \rightsquigarrow l_e, \Sigma^*, \phi', \Delta(l_e)}$$

$$T - ite\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \phi' \triangleq \phi \wedge \varphi_e = true}{l, \Sigma^*, \phi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_1, \Sigma^*, \phi', \Delta(l_1)} \quad F - ite\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \phi' \triangleq \phi \wedge \varphi_e = false}{l, \Sigma^*, \phi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_2, \Sigma^*, \phi', \Delta(l_2)}$$

$$var\ assign\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \Sigma^*_{new} \triangleq \Sigma^*[v \leftarrow fresh] \quad \phi' \triangleq \phi \wedge fresh = \varphi_e}{l, \Sigma^*, \phi, v := e \rightsquigarrow l+1, \Sigma^*_{new}, \phi', \Delta(l+1)}$$

$$@\ assign\ \frac{\Sigma^*, e \vdash_e \varphi \quad \Sigma^*, e' \vdash_e \varphi' \quad \Sigma^*_{new} \triangleq \Sigma^*[Mem \leftarrow fresh_m] \quad \phi' \triangleq \phi \wedge fresh_m = store(\Sigma^*(Mem), \varphi', \varphi)}{l, \Sigma^*, \phi, @\ e' := e \rightsquigarrow l+1, \Sigma^*_{new}, \phi', \Delta(l+1)}$$

$$stop\ \frac{}{l, \Sigma^*, \phi, stop \rightsquigarrow \mathbf{return}\ \phi}$$

Figure 2: Path predicate computation.

3. MOTIVATIONS

3.1 The Case for C/S Policy Specification

Let us consider an instruction `x := @(a*b)` (recall that `@` denotes a dereferencing operator), and a policy stating that read expressions should be *concretized*. Let us assume that runtime values are 7 for `a` and 3 for `b`. Then, there are at least three ways of understanding this "concretization" (M is a variable representing memory):

CS1: $x == select(M, 21)$ [incorrect]

CS2: $x == select(M, 21) \wedge a \times b == 21$ [minimal]

CS3: $x == select(M, 21) \wedge a == 7 \wedge b == 3$ [atomic]

The first formula is simple yet *incorrect* (we lose that $a \times b == 21$), the second formula is correct and *minimal* w.r.t. the initial objective, and the third formula is correct but very restrictive since it imposes values for both `a` and `b` — we can see it as a *atomic* concretization of read expressions, affecting only variables. Note however that **CS2** does not allow to get rid of the $\times$ operator, which may be a problem if our solver does not support it, while both **CS1** and **CS3** do. None of these three policies is clearly better, it all depends on the context. Yet, which policy was intended?

This example demonstrates that C/S policies can show subtle differences, and that clear specifications are required.

3.2 The Case for Dedicated C/S Policies

Let us consider the C program presented in Figure 3, where `x`, `y` and `z` are supposed to be tainted [26], i.e. on the control of the user. This program is therefore vulnerable: variable `buf` can be overflowed at lines 7 and 8. In both cases, the pointer `ptr` could be overwritten[3], potentially hijacking the function call at line 9.

[3]We assume here that `ptr` is just below `buf` in the stack frame, which is compiler-dependent.

Nevertheless, only the second case (line 8) is interesting from an exploitability point of view, since the attacker can control both the value to write (`z*2`) and the destination address (`buf[y]`), while in the first case value `42` probably does not produce an executable address.

```
1 void example1(int x,int y,int z) {
2   int val;
3   int buf[10];
4   void (*ptr)(void);
5   ptr=&fun1; //a function elsewhere
6   val=z*2; //val is tainted
7   buf[x]=42;//buf[x] tainted, 42 is not
8   buf[y]=val;//buf[y] and val tainted
9   ptr();
10 }
```

Figure 3: Motivating example

Let us suppose that our goal is to detect such form of weakness through DSE. Here, results will highly depend on the C/S choices on memory reads and writes. For instance, we can consider two standard C/S policies :

CS4: concretize the destination address of every write operation, propagate (i.e. keep symbolic) otherwise.

CS5: concretize the destination address of every write operation if not tainted, propagate otherwise.

CS4 scales well since it avoids the computation of array formulas with symbolic indexes (which introduce combinatorial reasoning), but it leads to a strong under-approximation of the path predicate, possibly preventing the detection of the vulnerability. On the other hand, since **CS5** keeps both `buf[x]` and `buf[y]` symbolic because they are tainted, the solver is indeed able to provide a valuation for `x` and `y` allowing to overwrite `ptr` and to hijack the control-flow. Yet, this policy keeps also `x` symbolic at line 7, leading to unnecessary computations here. A more appropriate policy would be to keep a write address symbolic only if *both* this address and the written value are tainted (policy **CS6**).

This example suggests that it is beneficial to tune C/S policies for specific application domains, as will be illustrated in Section 6.3.

4. SPECIFICATION OF C/S POLICIES

Our main goal is to design a high-level specification language for C/S policies able to encode the major policies found in the literature. Especially, we want to be able to distinguish between **CS1**, **CS2** and **CS3** (Section 3.1) and to express **CS4** and **CS5** (Section 3.2). Besides expressiveness, the following properties are also desirable: (1) a clear semantics; (2) simplicity and concision; (3) independence from the code under analysis or from a particular SE tool; (4) "executability", in that we want to synthesize the code enforcing a C/S policy from its specification.

We achieve these goals through CSML, a high-level *rule-based language* offering pattern matching and subterm checks on the expression and the instruction being processed. As an example, the encoding of **CS4** is presented in Table 2. It will be explained in Section 4.2.

$*$	::	$\langle @?e := ?\star\rangle$	::	$\langle !e\rangle$	::	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

Table 2: CS4 policy

Before presenting CSML together with its semantics (Section 4.2), we define formally what a C/S policy is, and how it interacts with path predicate computation (Section 4.1).

4.1 Revisiting DSE with C/S Policies

Formalization of C/S policies. The goal of a C/S policy is to decide whether a given expression of the execution trace must be:

- concretized ($\mathcal{C}$), i.e. replaced by its concrete value in the trace;
- symbolized ($\mathcal{S}$), i.e. replaced by a fresh (unconstrained) symbol;
- or propagated ($\mathcal{P}$), keeping its current value as in the standard algorithm of Figure 2.

We denote by $\rho \triangleq \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$ the set of possible decisions, and by $\mathbb{S}tate$ the set of concrete memory states. We define a C/S policy *csp_expr* as a function that takes as input a location l, an instruction i, a concrete memory state Σ and an expression e, and returns a decision $d \in \rho$. Formally:

$$csp\_expr : Loc \times \mathbb{I}nstr \times \mathbb{E}xpr \times \mathbb{S}tate \to \rho$$

Path predicate computation with C/S policy. The C/S policy is queried inside the path predicate computation algorithm each time an expression must be evaluated. Intuitively, the standard algorithm of Figure 2 corresponds to the case where the decision is always $\mathcal{P}$ (*propagate current value*), starting from an initial state where all variables are symbolized.

A version of the path predicate computation algorithm revisited for taking C/S policies into account is presented in Figure 4. The main difference with the standard approach is that the symbolic evaluation operator $\vdash_e$ is slightly modified into $\vdash_{cs}\bullet$ to use the policy. Moreover, before evaluating an expression, $csp\_expr(l, i, e, \Sigma)$ is queried in order to know which action shall be performed (lower part of Figure 4):

- $\mathcal{S}$: the expression e is replaced by a new symbol;
- $\mathcal{P}$: the expression e is symbolically evaluated, applying the C/S policy on each sub-terms [with $\vdash_{cs}\circ$];
- $\mathcal{C}$: the expression e is symbolically evaluated [with $\vdash_{cs}\circ$], then the resulting logical expression is constrained to be equal to the concrete evaluation $eval_\Sigma(e)$ of e, in order to preserve correctness of concretization.

Note that $\vdash_{cs}\bullet$ [and $\vdash_{cs}\circ$] returns now both a symbolic expression *and* a formula, the latter representing the constraints potentially induced by concretizing some subterms of the expression to evaluate.

4.2 Specification Language for C/S Policies

We describe now CSML, our rule-based language whose syntax is given in Table 3.

Basic principles. A rule is of the form *guard* $\Rightarrow \rho$ where the guard allows to check if the rule should be fired (typically using *pattern-matching* and *subterm checks*) and ρ is the decision to be returned by the policy. As explained before, rules are queried with the current location, instruction, expression and concrete memory state. Namely, guards are denoted by $\pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_\Sigma$, where:

- π_{loc} is a predicate on the location,
- π_{ins} is a predicate on the instruction,
- π_{expr} is a predicate on the expression,
- π_Σ is a predicate on the concrete memory state.

Rules are tested sequentially, and the first fireable rule returns its associated action. If no rule fires, then a default rule is applied. Inside each rule, guard predicates are also checked sequentially, so that π_{loc} must be satisfied in order to check π_{ins} and so on. Note that any of these predicates can be replaced by $*$, which always evaluate to true. Finally, guard predicates may communicate in a limited way through *meta-variables* and *placeholders*.

policy	::=	rules default $\mid$ default
rules	::=	rule $\mid$ rule rules
rule	::=	guard $\Rightarrow \rho$
default	::=	*default* $\Rightarrow \rho$
guard	::=	$\pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_\Sigma$
π_{loc}	::=	loc $\mid$ [loc..loc] $\mid$ *
π_{ins}	::=	$\langle pinstr \rangle$ $\mid$ *
π_{expr}	::=	$\langle pexpr \rangle$ $\mid$ $expr^+ \prec$ term$^+$ $\mid$ $expr^+ \preceq$ term$^+$ $\mid$ *
term$^+$	::=	$expr^+$ $\mid$ $instr^+$
π_Σ	::=	$P(expr)$

. $expr^+$: extended expression, allowing placeholders
. *pexpr*: expr. pattern, allowing placeholders and meta-variables
. $instr^+$ and *pinstr*: the same w.r.t. instructions

Table 3: Policy language

Matching and meta-variables. Predicates for π_{expr} and π_{ins} typically allow to check if the input expression (resp. instruction) matches a given pattern *pexpr* (resp. *pinstr*). A

$$\mathbb{E}\mathbf{xpr}: \quad cst\ \frac{}{\Sigma^*, bv \vdash_{cs^\circ} bv, true} \quad var\ \frac{}{\Sigma^*, v \vdash_{cs^\circ} \Sigma^*(v), true} \quad binop\ \frac{\Sigma^*, e_1 \vdash_{cs^\bullet} \varphi_1, \phi_1 \quad \Sigma^*, e_2 \vdash_{cs^\bullet} \varphi_2, \phi_2}{\Sigma^*, e_1 \diamond_b e_2 \vdash_{cs^\circ} \varphi_1 \diamond_b^* \varphi_2, \phi_1 \wedge \phi_2}$$

$$unaryop\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi_e, \phi_e \quad \varphi' \triangleq \diamond_u^* \varphi_e}{\Sigma^*, \diamond_u e \vdash_{cs^\circ} \varphi', \phi_e} \quad @\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi_e, \phi_e \quad \varphi \triangleq select(\Sigma^*(Mem), \varphi_e)}{\Sigma^*, @\ e \vdash_{cs^\circ} \varphi, \phi_e}$$

$$\mathbb{I}\mathbf{nstr}: \quad goto\ l_1\ \frac{}{l, \Sigma^*, \phi, goto\ l_1 \rightsquigarrow l_1, \Sigma^*, \phi, \Delta(l_1)} \quad l_e - goto\ e\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi_e, \phi_e \quad \phi' \triangleq (\phi \wedge \phi_e \wedge to\_val(l_e) = \varphi_e)}{l, \Sigma^*, \phi, goto\ e \rightsquigarrow l_e, \Sigma^*, \phi', \Delta(l_e)}$$

$$T - ite\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi_e, \phi_e \quad \phi' \triangleq \phi \wedge \phi_e \wedge \varphi_e}{l, \Sigma^*, \phi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_1, \Sigma^*, \phi', \Delta(l_1)} \quad F - ite\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi_e, \phi_e \quad \phi' \triangleq \phi \wedge \phi_e \wedge \neg\varphi_e}{l, \Sigma^*, \phi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_2, \Sigma^*, \phi', \Delta(l_2)}$$

$$var\ assign\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi_e, \phi_e \quad \Sigma^*_{new} \triangleq \Sigma^*[v \leftarrow fresh] \quad \phi' \triangleq (\phi \wedge \phi_e \wedge fresh = \varphi_e)}{l, \Sigma^*, \phi, v := e \rightsquigarrow l+1, \Sigma^*_{new}, \phi', \Delta(l+1)}$$

$$@\ assign \frac{\Sigma^*, e \vdash_{cs^\bullet} \varphi, \phi_e \quad \Sigma^*, e' \vdash_{cs^\bullet} \varphi', \phi_{e'} \quad m' \triangleq store(\Sigma^*(Mem), \varphi', \varphi) \quad \phi_m \triangleq (\phi \wedge \phi_e \wedge \phi_{e'} \wedge fresh_m = m')}{l, \Sigma^*, \phi, @\ e' := e \rightsquigarrow l+1, \Sigma^*[Mem \leftarrow fresh_m], \phi_m, \Delta(l+1)}$$

$$\Sigma^*, e \vdash_{cs^\bullet}: \left\{ \begin{array}{lll} fresh, true & \text{if} & \rho = \mathcal{S} \\ \varphi_e, \phi_e & \text{if} & \rho = \mathcal{P}, \quad \Sigma^*, e \vdash_{cs^\circ} \varphi_e, \phi_e \\ C_\varphi, \phi_e \wedge C_\varphi = \varphi_e & \text{if} & \rho = \mathcal{C}, \quad \Sigma^*, e \vdash_{cs^\circ} \varphi_e, \phi_e \text{ and } C_\varphi \triangleq eval_\Sigma(e) \end{array} \right\} \rho \triangleq csp\_expr(l, i, e, \Sigma)$$

Instruction, location l and concrete state Σ are propagated inside all $\vdash_{cs^\bullet}$ rules, but we omit it for clarity.

Figure 4: Path predicate computation with C/S policy

pattern is similar to an expression (resp. instruction), but with two additional kinds of variables. *Meta-variables* (prefixed by ?) allow to match any term. Once successfully matched, these terms become available as placeholders in the subsequent guard predicates. When a meta-variable does not need to be reused, one can employ an *anonymous meta-variable* $?\star$. *Placeholders* (prefixed by !) take the value of their corresponding meta-variable (e.g., $?e$ for $!e$) once matched. The distinguished placeholder $!_\Box$ contains the current expression being processed. Given a pattern p, we denote the predicate "match p?" by $\langle p \rangle$. As an example, we concretize the expression being added to `esp`, in any assignment instruction:

$$* \ :: \ \langle ?\star \ := \texttt{esp + } ?e \ \rangle :: \ \langle !e \ \rangle :: \ * \ \Rightarrow \ \mathcal{C}$$

Here, $?e$ is defined in π_{ins} and then available in π_{expr} through $!e$. The rule reads as follows: if the current instruction assigns the sum of `esp` and some expression e to any lhs and the current expression matches e, then it should be concretized. Or, put another way: if we are evaluating an expression e in the context of an instruction where e is added to `esp` and assigned to some lhs, then e should be concretized.

We can now understand the encoding of **CS4** given in Table 2: if we are evaluating an expression e in the context of an assignment where e is used as the write address, then e is concretized, otherwise it is propagated.

Subterm. This language is already quite expressive, but still does not allow to match a nested sub-expression depending on its context. The typical usage is, when willing to concretize a read address, we want to know if the given expression is in the scope of a read operation. This can be solved by introducing the $\preceq$ operator (resp. $\prec$), allowing to check if an expression is a subterm (resp. strict subterm) of another one. Specifying that any read or write expression must be concretized can then be simply written as follows, where $?i$ matches a whole instruction:

- correct concretization of r/w expressions [**CS2**]

$$* \ :: \ \langle ?i \ \rangle :: \ (@\ !_\Box) \prec !i :: \ * \ \Rightarrow \ \mathcal{C}$$

The rule can be read as follows: when evaluating an expression e (given by the special placeholder $!_\Box$) such that the expression $@e$ is a subterm of the current instruction (captured by $?i$) then e should be concretized.

Note that $\prec$ and $\preceq$ can be applied to (extended) terms containing placeholders ($expr^+$ and $instr^+$ in Table 3).

Other predicates. π_{loc} consists in checking that the input location l is either equal to a given location loc or within a range of locations [loc..loc']. π_Σ is a predicate over $\mathbb{E}xpr$ that can query information from Σ. For example, we can imagine performing some concretization and/or symbolization depending of the runtime value of the expression being evaluated. The interest of π_Σ will be lightened once we allow *extended memory states* (cf. Section 4.4).

4.3 Properties of the Specification Language

CSml ensures interesting properties on the C/S policy being defined. In order to present them, we need a few additional definitions. A CSml rule is said to be *well-defined* if it is well-typed and place holders are used in an appropriate manner. Note that the well-definedness of a rule is automatically checkable. A C/S policy is *well-defined* if it is a total function (deterministic behavior, defined on any input), and it is *correct* if it yields to the computation of a correct path predicate (cf. Section 2.3). Then, by construction, we have the good following properties[4]:

Property 2. *A set of well-defined* CSml *rules defines a well-defined C/S policy.*

[4]Property 2 comes also from the sequential ordering of rules and the presence of a default rule.

Property 3. *A set of well-defined* CSml *rules employing only $\mathcal{C}$ and $\mathcal{P}$ defines a correct C/S policy.*

4.4 Advanced Features

We propose here a few extensions to the CSml core language, presented with less details due to space limitations.

Richer decisions. We enrich the set of possible decisions by allowing a domain restriction on both $\mathcal{S}$ and $\mathcal{P}$, now denoted $\mathcal{S}_D$ and $\mathcal{P}_D$, where D is an interval constraint (resp. singleton constraint) of the form $[a..b]$ (resp. $[a]$) with a and b expressions evaluating to bitvector values. This feature is useful typically for limiting the domain of a fresh logical variable, but it can also be used to encode incorrect concretization or restricted propagation (cf. below). Note that the domain does not need to be defined by constant values, for example its definition can involved runtime evaluation of bitvector expressions, through function $eval_{\Sigma} : \mathbb{E}xpr \rightarrow Val$

- incorrect concretization of r/w expressions [**CS1**] [24]
$$* \ :: \ \langle ?i \ \rangle \ :: \ (@ \ !_{\square}) \prec !i \ :: \ * \ \Rightarrow \ \mathcal{S}_{[eval_{\Sigma}(!_{\square})]}$$
- restriction of r/w expressions [5]
$$* \ :: \ \langle ?i \ \rangle \ :: \ (@ \ !_{\square}) \prec !i \ :: \ * \ \Rightarrow \ \mathcal{P}_{[eval_{\Sigma}(!_{\square})-10..eval_{\Sigma}(!_{\square})+10]}$$

Richer subterm constraints. We allow chaining of subterm constraints, e.g. $e \prec pe_1 \prec \ldots \prec pe_n \prec \text{term}^+$, together with the use of anonymous meta-variables inside the pe_i. This allows finer subterm relationship, such as checking that an expression is a subterm of an expression pattern, used itself within another expression.

- recursive concretization of r/w expressions:
$$* \ :: \ \langle ?i \ \rangle \ :: \ !_{\square} \prec (@ \ ?\star) \prec !i \ :: \ * \ \Rightarrow \ \mathcal{C}$$

Compared with concretization **CS2** presented in Section 4.2, this rule enforces the concretization of all subterms of a r/w expression. This policy is also slightly different from atomic concretization **CS3**, whose encoding is shown hereafter.

Richer predicates. We can also allow more predicates in the language. This can be done at two stages: either enriching the four classes of predicates already defined, or adding new classes of predicates. For the first category, it could be useful to have a `var` predicate indicating if a term is a variable or not. An application is to restrict concretization to atomic variables:

- atomic concretization of r/w expressions [**CS3**]
$$* \ :: \ \langle ?i \ \rangle \ :: \ \texttt{var}(!_{\square}) \land !_{\square} \prec (@ \ ?\star) \prec !i \ :: \ * \ \Rightarrow \ \mathcal{C}$$

For the second category, it could be useful to consider a predicate class π_{step} regarding the step of the execution, allowing for example to define a C/S policy step by step in a trace-oriented manner, which may sometimes come handy.

Extended memory states. In the same vein, it can be interesting to enrich the predicate π_{Σ} working on a concrete memory state Σ into a predicate π_{Σ^+} working on an extended concrete memory state Σ^+. Basically, an extended concrete memory state is a concrete memory state enriched with additional runtime information collected. A typical example of such a predicate is $\widehat{\mathcal{T}}_{\Sigma^+}(e)$, indicating whether an expression e is tainted or not in a given extended memory state Σ^+, using dynamic taint information [26].

Note that this extension makes the C/S policy dependent on the services provided by the underlying dynamic execution engine. While it is fair to assume that a concrete evaluation function $eval_{\Sigma}$ is available on any dynamic execution engine, more exotic queries on Σ^+ may not be available. We assume that C/S policies querying unsupported Σ^+-function (or Σ^+-predicate) are syntactically rejected.

C/S through memory injection. Besides C/S at the level of symbolic evaluation, another common pattern is to enforce concretization and/or symbolization through direct modification of the symbolic memory state. This is particularly useful to handle unknown or hard-to-reason-about functions (e.g. system calls, cryptographic function) with side-effects or returning complex data structures. Note that this kind of C/S is different from the one we have considered so far, since it modifies *permanently* the value of a *lhs* (inside Σ^+), while *csp_expr* affects a *single evaluation* of *any expression*. For example, C/S memory injection allows to declare that at some location, variable `eax` receives a fresh value (which will last along the trace until `eax` is rewritten), while *csp_expr* allows to declare that at some location, variable `eax` evaluates as if it were unconstrained (with no impact on the remainder part of the trace).

C/S injection can be handled similarly to C/S in expression evaluation. Due to space limitation, we only sketch the idea. We introduce a new function

$$csp\_mem : Loc \times \mathbb{I}nstr \times \mathbb{S}tate \rightarrow (lhs \mapsto \{\mathcal{C}, \mathcal{S}\})$$

which takes as argument a location, an instruction and a memory state, and returns a map from *lhs* to decisions, which are here limited to $\mathcal{C}$ and $\mathcal{S}$. Intuitively, the map represents the modifications which have to be performed on the current symbolic memory state Σ^* before the symbolic execution goes on.

Contrary to *csp_expr*, concretizations defined by *csp_mem* are not ensured to be correct, as the symbolic memory state is modified without any additional (correctness) constraint.

Discussion. Altogether, these extensions provide a very fine control over the C/S policy, allowing for example to encode the subtle differences between correct concretization, incorrect concretization, recursive concretization and atomic concretization.

4.5 Encoding Standard C/S Policies

To illustrate how our language works, we show the encoding of several state-of-the-art policies from the literature, not yet covered in Sections 4.2 and 4.4.

For instance, CUTE and DART [27, 22] concretizes both read and write addresses, as well as part of non-linear operations (here: left operand of any $\times$ operator). The associated policy is shown in Table 4.

$*$	$::$	$\langle ?i \rangle$	$::$	$(@ \ !_{\square}) \prec !i$	$::$	$*$	$\Rightarrow \mathcal{C}$;
$*$	$::$	$\langle ?i \rangle$	$::$	$(!_{\square} \times ?\star) \prec !i$	$::$	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

Table 4: CUTE/DART policy

A variant for memory operations consists in concretizing also all non-tainted expressions [24]. The corresponding policy is shown in Table 5, where $\widehat{\mathcal{T}}_{\Sigma^+}(e)$ indicates whether an expression e is tainted or not in a given extended memory state Σ^+ (cf. Section 4.4).

$*$	::	$\langle ?i \rangle$	::	$(@\ !_{\square}) \prec\ !i$	::	$*$		$\Rightarrow \mathcal{C}$;
$*$	::	$*$	::	$*$	::	$\neg\widehat{\mathcal{T}}_{\Sigma^+}(!_{\square})$		$\Rightarrow \mathcal{C}$;
default								$\Rightarrow \mathcal{P}$;

Table 5: CUTE/DART policy with tainting

The approach followed in EXE [13] in case of multi-level dereferencement consists in concretizing all r/w expressions but the most nested one. The encoding of such a policy is shown in Table 6.

$*$	::	$\langle ?i \rangle$	::	$(@\ !_{\square}) \prec (@\ ?\star) \prec\ !i$	::	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

It is important here to use $\prec$ rather than $\preceq$.

Table 6: EXE policy

Finally, the policy in Mayhem [17] consists in concretizing all write expressions while keeping read expressions symbolic as long as they cannot take too many values (otherwise, concretizing them). We need here to consider Σ^+ enriched with an interval analysis. The encoding is then given in Table 7, where $card_I(e)$ gathers the number of possible values for e from the interval information available in Σ^+.

$*$	::	$\langle @?a\ := ?\star \rangle$	::	$\langle !a \rangle$	::	$*$	$\Rightarrow \mathcal{C}$;
$*$	::	$\langle ?i \rangle$	::	$(@!_{\square}) \prec\ !i$	::	$card_I(!_{\square}) < 1024$	$\Rightarrow \mathcal{P}$;
$*$	::	$\langle ?i \rangle$	::	$(@!_{\square}) \prec\ !i$	::	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

Table 7: Mayhem policy

Summary. Table 8 presents a summary of the kinds of C/S policies CSML can encode, together with the required extension of the language. It is remarkable that CSML can encode all popular C/S policies despite a limited language. Hence, we think that our rule-based language manage to capture the crucial aspects of current C/S policies.

Limits. We do not know of any major existing C/S policy that cannot be encoded into CSML. Yet, the framework has some limitations, coming from both the ordered evaluation of guard predicates and the very restricted communication between those predicates. Here are two such limitations.

- A C/S policy does not depend on the symbolic state we are building, for example we cannot decide to concretize a term if all its leaves (variables) are already concretized.

- A C/S policy does not depend on the formula we are solving. For example, we cannot compute a path predicate, pass it to a solver (or to a simplifier) and then request concretization or symbolization depending on the solver's output.

Note, however, that the extended memory state Σ^+ does allow to overcome most of the above limitations, assuming one is willing to store (resp. to query) very complex information into (resp. from) Σ^+. In our view, Σ^+ should be used with care, only in last resort.

policy		language
minimal concretization	[**CS2**]	core language
recursive concretization		extended $\prec$
atomic concretization	[**CS3**]	extended $\prec$ and `var`
incorrect concretization	[**CS1**]	extended decisions
r/w full-concrete	[dart/cute]	core language
r/w full-symbolic	[pathcrawler]	core language
r/w domain restriction	[osmose]	extended decisions
r/w multi-level	[exe]	extended $\prec$
r/w taint-based	[24]	extended Σ^+
r/w dataflow-based	[mayhem]	extended Σ^+

Table 8: Encoding of C/S policies

5. IMPLEMENTATION

The C/S policy mechanism presented so far has been integrated into BINSEC/SE [20], an open-source DSE tool built on top of the BINSEC framework [21] for binary code analysis. BINSEC and BINSEC/SE are developed in OCaml. They rely on DBA [6] and use solvers Z3 [8] and Boolector [11].

An overview of the modified architecture is shown in Figure 5. The C/S policy is specified in a textual format close to CSML. Subsequently, while the SE engine creates the path predicate, the C/S policy is queried for each encountered expression (Section 4.1) via a hook function, instantiated from the CSML specification.

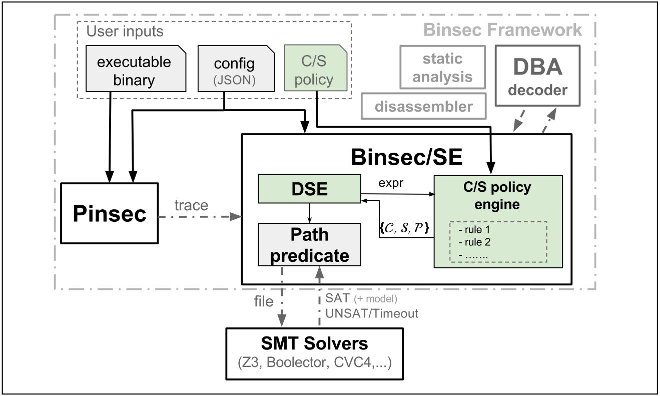


Figure 5: CSML support in BINSEC/SE

This version of BINSEC/SE is currently *the first SE tool supporting high-level specification of a wide range of C/S policies.* The core engine is fully functional: all experiments of Section 6 have been carried out with it. Concerning CSML, the whole core language is supported, as well as extended memory states (currently: taint and heap information) and extended decisions. Other features are in progress, especially memory injection.

6. EXPERIMENTS

We report in this section three experiments that have been carried out with CSML and BINSEC/SE.

- First, we are interested in studying the impact of C/S policies targeting memory reads and writes. While (the new) policy PP* works better on average, a generic C/S mechanism is still strongly recommended. This is the first time such a comparison is performed.

- Second, we are interested in estimating the overhead of rule-based C/S specification. The conclusion is that our approach does impact the cost of formula creation, yet it is still negligible w.r.t. formula solving. Hence, our approach is practical.
- Finally, we come back to the motivating example of Section 3.2, in order to argue on the benefit of defining specific purpose-oriented C/S policies.

6.1 Quantitative Evaluation of C/S Policies

We study the impact on SE of C/S policies targeting memory reads and writes. This is a typical application of C/S, since a faithful modeling of memory operations may lead to hard-to-solve formulas. More precisely, we investigate the following questions: **RQ 1**: Do C/S policies have a significant impact on SE in terms of the quality of results? **RQ 2**: Is there a best C/S policy for read/write operations – among standard policies?

Protocol. We consider 5 different C/S policies regarding the handling of memory reads and writes, namely: CC, CP, PC, PP*, PP, where the first letter indicates whether read addresses are concretized (C) or propagated (P), and the second letter indicates the same for write addresses. PP* is a special case: all read and write addresses are kept symbolic, except for stack registers (i.e., on x86, registers `esp` and `ebp` are concretized). While CC and PP are standard policies, the three others are new.

Experiments are performed over 167 programs (x86 executable codes) – composed of programs from NIST/SAMATE [30] (a standard benchmark for program analysis), all Unix coreutils and several Windows malware [31], for a total of 45,242 solver queries. Details can be found in Table 9. All instructions are traced, except calls to library functions which are stubbed by symbolic values (fresh logical variables). The solver is Z3, with a time-out of 30 seconds.

Table 9: Benchmark characteristics

category	#prog	# trace instr		# trace branch		
		max	avg	max	avg	total
samate	50	5,000	2,772	1,177	333	16,554
coreutils	100	5,000	1,572	1,053	171	17,198
malware	17	5,000	3,739	1,539	675	11,490
all	167	5,000	2,151	1,539	270	45,242

We measure the influence of these policies in the following way: for each benchmark program, we consider an arbitrary (but reproducible) initial concrete execution and we ask the SE engine to iteratively invert every condition along the initial execution, leading to a set of new path predicate computations and solver queries. We record for each policy the number of queries which have been successfully solved (SAT), proved infeasible (UNSAT) or which have triggered a time-out (TO). Note that only the first category leads to new test input and (hopefully) better code coverage.

Results and conclusion. Part of results[5] are summarized in Tables 10 and 11. First, [**RQ 1**] the choice of C/S policy may greatly affect the outcome of SE: there are $\geq$5x more SAT results on 20/167 examples, and up to 286x more SAT results on one program. Second, [**RQ 2**] there is no clear hierarchy between the considered policies. Indeed, even if PP* performs very well on many examples — PP* is the best policy on 41/167 examples, and it is optimal on 117/167 examples (Table 11), the global number of successfully solved instances is pretty close for all policies but PP (Table 10). Actually, while a more symbolic policy leads in theory to more satisfiable queries, it may also come at the price of harder-to-solve formulas and time-outs. These results are a strong argument in favor of a generic C/S mechanism.

[5]All our data will be made publicly available.

The major *threats to validity* are the representativeness of the experimental setting (policies, programs) and internal bugs in the SE tool. We mitigate these threats through using standard policies and variants of them as well as a large program set coming from three distinct well-known and publicly-available benchmarks. Moreover, we rely on publicly-available tools (SE, solver) and results have been crosschecked for internal validity.

Table 10: Summary of #SAT, #UNSAT and #TO

	#SAT	#UNSAT	#TO
CC	4,518	40,712	12
PC	4,436	38,897	1,909
CP	4,651	39,310	1,281
PP*	4,515	31,320	9,407
PP	3,340	25,037	16,865

total number of queries: 45,242

Table 11: Best and optimal policies

	samate		core		malware		total	
	opt	best	opt	best	opt	best	opt	best
CC	20	0	44	1	5	0	69	1
PC	20	2	49	4	6	1	75	7
CP	23	1	61	11	4	0	88	12
PP*	36	12	71	24	10	5	117	41
PP	33	9	36	7	7	2	76	18

total number of programs: 167 - best (resp. opt): number of programs for which the considered policy returns the strictly highest (resp. highest) number of SAT answers, w.r.t. the other policies.

6.2 Overhead of the Rule-based Language

We evaluate the overhead of our parametric C/S policy mechanism. We want to answer the two following questions: **RQ 3:** What is the extra-cost of rule-based C/S specification, especially w.r.t. hard-coded policies (by mean of callback functions) and no C/S policy at all? **RQ 4:** Is it affordable, i.e. is the extra-cost low w.r.t. solving time?

Protocol. We reuse the experimental setting of the previous evaluation. We consider two metrics: the cost of formula creation – which is directly affected by C/S policies, and the ratio between formula creation and formula solving. We record these metrics for the 5 previous policies, implemented either in CSML or through native callbacks, and we consider a baseline consisting of SE without any C/S policy.

Results and conclusion. Table 12 reports the ratio between formula creation and formula creation plus solving. Note that solving time does not depend on the way C/S is implemented. [**RQ 3**] CSML does lead to a more expensive path predicate computation (average: x3 w.r.t. hard-coded callbacks and up to x5 w.r.t. no C/S at all, at worst x7 on some examples), yet [**RQ 4**] the cost of predicate computation is still negligible (average of 1.45% for the most expensive C/S policy; maximum of 23% on some easy-to-solve path predicates) w.r.t. the cost of predicate solving. Hence, our rule-based C/S mechanism brings extra-flexibility at only a very slight extra-cost.

Threats to validity: besides issues discussed in the previous evaluation, the considered policies are rather simple w.r.t. the expressive power of CSml. While the study is of interest because these policies are representative, further investigations are required for *complex* CSml policies.

Table 12: Overhead evaluation

		min	max	average
base	(PP)	0.04%	3%	0.3%
rule-based C/S policy	CC	0.1%	17%	1.2%
	CP	0.1%	23.5%	1.45%
	PC	0.08%	12.8%	0.85%
	PP*	0.08%	12.3%	0.95%
	PP	0.05%	4%	0.48%
hard-coded C/S policy	CC	0.05%	8.5%	0.5%
	CP	0.05%	8.2%	0.5%
	PC	0.05%	8%	0.45%
	PP*	0.05%	6%	0.45%
	PP	0.04%	3%	0.3%

Ratio between the cost of path predicate computation (impacted by C/S) and the whole cost (i.e. formula creation + formula solving). Note that the time for formula solving does not depend on the way C/S is implemented (rules, hard-coded, no C/S).

6.3 Dedicated C/S Policies

We come back to the motivating example of Figure 3. In order to check that vulnerability at line 9 can be exploited, we follow the general line of [1, 24, 17] and strengthen the path predicate with the extra condition that at line 9, `ptr` must be equal to an arbitrarily-chosen value, here `0x61626364`. Depending on C/S, our strengthened path predicate may or not be satisfiable. We consider **CS4** and **CS5** (Section 3.2), as well as the (original) following one:

CS6: Memory writes are kept symbolic if *both* the destination address and the value to write are tainted.

The encoding of **CS4** is given in Table 2, those of **CS5** and **CS6** are straightforward from **CS4** and Table 5. As expected, **CS6** does allow to recover input values *triggering the exploit and hijacking the execution control flow* to address `0x61626364` (with Z3: `x=0x0`, `y=0xb` and `z=0x30b131b2`), while **CS4** *does not reveal the exploit* (the formula is too constrained), and **CS5** obtain an exploit, but *with a more complex formula* (with an extra symbolic expression for `&buf[x]`).

7. RELATED WORK

Several DSE frameworks have been developed so far, each of them offering its own solution to the C/S issue. We summarize hereafter the most representative solutions, and we compare them with our approach.

Built-in C/S policies. Most SE tools implement a single hard-coded built-in C/S policy, which can favor either *scalability* (i.e., by considering most values as concrete) or *completeness* (i.e., by keeping more symbolic values). For instance, the pioneering tools DART [22] and CUTE [27] fall in the former category (memory addresses, results from external library calls and part of non-linear expressions are concretized), while PATHCRAWLER keeps the computation fully symbolic [29] and EXE [13] stands in between. More recent engines can even build on more sophisticated heuristics in the hope of reaching a sweet spot between scalability and completeness, typically based on tainting [26, 24] or dataflow analysis [17]. We showed in Section 4.5 how such policies can be specified in CSml.

More flexible policies. Klee [12] is a popular symbolic execution engine operating on LLVM byte-code [15]. By default, each program variable is considered as concrete, unless specified otherwise. Source-level primitives [6] allows to indicate that a variable should be considered as "symbolic" from a given control location. Finally, symbolic values are (implicitly) made concrete when calling native external libraries, unless a model (i.e. a C stub) is provided.

S2E [19, 18] allows to perform symbolic execution at *system level*, taking into account not only the target application but also its whole environment. Although partially based upon Klee, S2E offers several original features, especially: a powerful – but rather complex – *plugin* mechanism allowing the user to interact with the execution engine by inserting external code upon reception of some events, and the ability to switch between symbolic and concrete modes. In addition, S2E provides several ways to introduce symbolic values at arbitrary memory locations.

Comparison with our proposal. Both S2E and Klee do allow the user to write and integrate C/S policies based on memory injection (cf. Section 4.4). They do not provide policies based on expression evaluation, while we argued in Section 4.4 that both notions are useful and orthogonal. Expressing a C/S policy with Klee is rather tedious and error-prone since such a policy should be explicitly and manually weaved into the code under test, S2E is more flexible, thanks to its plugin mechanism.

These two approaches focus on practical implementation, while we are also interested in the formalization, specification and (ultimately) understanding of C/S policies. Especially, we propose a much more comprehensive and declarative way to define C/S policies.

8. CONCLUSION

Concretization and Symbolizations (C/S) is a crucial part of modern SE tools, yet C/S is often treated as "black magic", with only little documentation and hard-coded heuristics. We propose a clear separation of concerns between the specification of C/S policies on one side, through the CSml rule-based language, and the algorithmic core of SE on the other side, revisited to take C/S policies into account. CSml is simple, yet powerful enough to encode all popular C/S policies from the literature. Such a mechanism has been implemented on top of Binsec/se, yielding the first SE tool supporting high-level specification of a wide range of C/S policies. We also carried out the first quantitative comparison on C/S policies, demonstrating that the level of genericity we offer is both very beneficial to SE and affordable (very low overhead). This work paves the way for a systematic study of C/S policies in order to better understand their impact and to identify interesting trade offs.

[6]like `klee_make_symbolic`

9. REFERENCES

[1] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Commun. ACM*, 57(2), 2014.

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with VeriTesting. In: ICSE 2014. ACM, 2014

[3] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbé. Binary-level testing of embedded programs. In: QSIC 2013. IEEE, 2013

[4] S. Bardin and P. Herrmann. Structural Testing of Executables. In: ICST 2008. IEEE, 2008

[5] S. Bardin and P. Herrmann. Osmose: Automatic structural testing of executables. *Software Testing, Verification Reliability*, 21(1), 2011

[6] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA framework for binary code analysis. In: CAV 2011. Springer, 2011

[7] S. Bardin, P. Herrmann, F. Védrine. Refinement-based CFG reconstruction from unstructured programs. In: VMCAI 2011. Springer, 2011

[8] N. Bjørner. Engineering theories with Z3. In: IWIL 2012, 2012

[9] E. Bounimova, P. Godefroid, and D. A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In: ICSE 2013. IEEE, 2013

[10] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries. Technical report, CMU-CS-07-133, 2007

[11] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In: TACAS 2009. Springer, 2009

[12] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008. USENIX Association, 2008

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In: CCS 2006. ACM, 2006

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In: ICSE 2011. ACM, 2011

[15] http://llvm.org

[16] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2), 2013

[17] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In: SP 2012. IEEE, 2012

[18] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In: ASPLOS XVI. ACM, 2011

[19] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1), 2012

[20] R. David, S. Bardin, T. Thanh Dinh, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In: SANER 2016. IEEE, 2016

[21] A. Djoudi and S. Bardin. BINSEC: Binary code analysis with low-level regions. In: TACAS 2015. Springer, 2015

[22] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6), 2005

[23] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012

[24] S. Heelan and A. Gianni. Augmenting vulnerability analysis of binary code. In: ACSAC 2012. ACM, 2012

[25] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976

[26] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In: SP 2010. IEEE, 2010

[27] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5), 2005

[28] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In: ICISS 2008. Springer, 2008

[29] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for C functions. In: ASE 2004. IEEE, 2004

[30] http://samate.nist.gov/

[31] http://vxheaven.org/