Data Minimisation: a Language-Based Approach*

Thibaud Antignac[1], David Sands[1], and Gerardo Schneider[2]

[1] Chalmers University of Technology, Sweden.
{thibaud.antignac,dave}@chalmers.se
[2] University of Gothenburg, Sweden.
gerardo@cse.gu.se

Abstract. Data minimisation is a privacy-enhancing principle considered as one of the pillars of personal data regulations. This principle dictates that personal data collected should be no more than necessary for the specific purpose consented by the user. In this paper we study data minimisation from a programming language perspective. We define a data minimiser as a pre-processor for the input which reduces the amount of information available to the program without compromising its functionality. We give its formal definition and provide a procedure to synthesise a correct data minimiser for a given program.

1 Introduction

According to the Article 5 of the EU *General Data Protection Regulation* proposal "Personal data must be [...] limited to what is necessary in relation to the purposes for which they are processed" [12]. This principle is called *data minimisation*. From a software perspective, data minimisation requires that the input data not *semantically* used by a program should neither be collected nor processed. The *data processor* could be seen in this context as to be the *adversary* (or *attacker*), as she knows all the information available after the input is collected (before the program execution) and thus can exploit the inputs.[3]

The goal of the *data minimisation process* is to minimise the input data so only what is necessary is given to the program. Whenever the input data exactly matches what is necessary we may say that the minimisation is the *best*. Best minimisation is, however, difficult to achieve in general since it is not trivial to exactly determine what is the input needed to compute each possible output. As an example of minimisation let us consider the program P_{bl} shown in Fig. 1, whose purpose is to compute the *benefit level* of employees depending on their salary (that we assume to be between \$0 and \$100000). For the

```
1 input(salary)
2 benefits := (salary < 10000)
3 output(benefits)
```

Fig. 1: Program P_{bl} computes benefit level.

* Published in *32nd Int. Conf. on ICT Systems Security and Privacy Protection - IFIP SEC'17*, vol. 502 of IFIP Advances in Inf. and Com. Tech. (AICT), pp. 442–456. Springer, May 2017. DOI: 10.1007/978-3-319-58469-0_30.

[3] In other scenarios the adversary only has access to the outputs (cf. [28]).

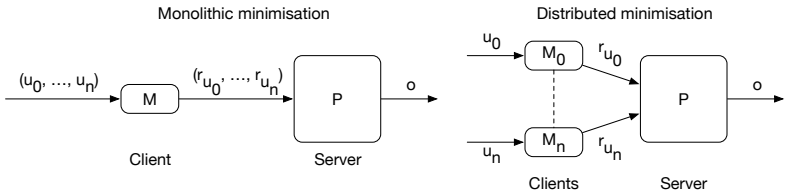

Fig. 2: Monolithic and distributed minimisation architectures.

sake of simplicity, in what follows we
drive our analysis on worst-case assumptions. A quick analysis shows that the range of the output is $\{\texttt{false}, \texttt{true}\}$, and consequently the data processor does not need to precisely know the real salaries to determine the benefit level. Each employee should be able to give any number between 0 and 9999 as input if eligible to the benefits, and any number between 10000 and 100000 otherwise, without disclosing real salaries.

In other words, data minimisation is the process of ensuring that the range of inputs provided to a program is reduced such that when two inputs result in the same response, then one of them can be considered redundant. Minimality is an information-flow property related to the notions of *required information release* [9], and of *strong dependency* (a.k.a. *non-interference*) [10]. Minimality can also be seen as one relating *cardinalities*: ideally, a program satisfying data minimisation should be one such that the cardinality of the possible output is equal to the cardinality of the input domain.

In a distributed setting the concept of minimisation is more complex since the minimisation process is not *monolithic* anymore as the data may be collected from multiple independent sources (users). Fig. 2 gives an illustration of this difference in the case of a static minimiser run by the client.

We will see that in general we cannot decide whether a program is minimal, or compute in the general case a "best" minimiser. In order to statically compute a minimiser for a program we need to compute its *kernel*. In the monolithic case this is just a partition of the input domain so that all the inputs in a partition get mapped to the same output. A minimiser in this case can be constructed by choosing a representative for each partition, and mapping all elements in that partition to the chosen representative. The distributed case is more complex. Though producing the coarsest possible partition is uncomputable, in practice it may be computed for specific cases. This is true for programs not having complex loops, mutual recursive calls, nor using libraries for which we have neither access to the code nor the specification. Our analysis does not completely exclude such programs as it might still be done in practice if suitable *invariants* are given, by providing formal specifications about libraries, or by sacrificing complete automation and allowing a man-in-the-loop to guide the analysis.

In this paper we address *data minimisation* with the following contributions: i) We provide a formal definition of the concept of *minimisers for a program* used to achieve data minimisation (Section 3) with respect to an explicit attacker

model (Section 2); ii) We show how we can compare minimisers, leading to the definition of *best minimiser* for a given program (Section 4); iii) We propose a (semi-)procedure to generate minimisers and prove its soundness and termination (Section 5); iv) We provide a proof-of-concept implementation (*DataMin*) to compute minimisers, based on symbolic execution and the use of a SAT solver in order to exemplify our approach.

The accompanying technical report [1] contains an application of our proof of concept to different prototypical programs, as well as more detailed examples and the proofs of the theorems and lemmas presented in the rest of the paper.

2 Attacker Model

In order to define data minimisation we consider an explicit attacker model. In practice a malicious server may have a *secondary use* for the data (defined as "using personal data in a different way than the primary reason why they were collected" [8]). We define here an attacker model having the following components: (i) an explicit model of a hidden secondary use of the data, and (ii) an explicit model of what the attacker learns about the data from observing the result of processing the data. We will develop a definition of minimisation which guarantees that an attacker learns no more about the private data than he can deduce from the legitimate output of the program.

To model the hidden secondary use, we suppose the attacker uses a second program denoted by a function $h \in \mathcal{I} \to \mathcal{O}'$. Thus the attacker is defined by a pair of functions p and h (the legitimate use and the hidden use of the data, respectively), and the attacker's computation is given by the function $\langle p, h\rangle \in \mathcal{I} \to \mathcal{O} \times \mathcal{O}'$ defined by $\langle p, h\rangle(i) \stackrel{\text{def}}{=} (p(i), h(i))$.

If the goal of the attacker is to learn something about the inputs by observing the output of $\langle p, h\rangle$ then it is easy to see that the worst case is when h is the identity function. In the following section we will show that if the input is first minimised using a best possible minimiser m, then what the attacker learns from $\langle p, h\rangle \circ m$ is no more than what is learned by p alone (with $\circ$ being the standard function composition operator). Here we assume that the attacker knows the minimiser m. To do this we model the attacker knowledge explicitly, as has become more common recently (e.g. [4, 5, 7]).

Definition 1 (Knowledge). *Let $u \in \mathcal{I}$ be an input, and f a function in $\mathcal{I} \to \mathcal{O}$. The* knowledge *about u obtained through f by observing $f(u)$, written $K(f, u)$, is defined to be the set $\{v \mid f(u) = f(v), v \in \mathcal{I}\}$.*

Thus $K(f, u)$ is the largest set of possible input values that could have led to the output value $f(u)$. This corresponds to an attacker who knows the function f and the output $f(u)$ and makes perfect logical deduction about the possible values of the input. Note that the smaller the knowledge set, the less uncertainty the observer has over the possible value of the input. The knowledge relation induces an order on programs: the bigger the knowledge set for a given input, the less a program will disclose information about the input. This is reasonable

given that if more input values are mapped to the same output, more difficult it will be to guess what is the exact input when observing the output of a computation. We have then the following order between programs:

Definition 2 (Disclosure ordering). *We say f* discloses less or as much information as *g, written $f \sqsubseteq g$, iff for all $u \in \mathcal{I}$ we have $K(g,u) \subseteq K(f,u)$.*

If f and g are knowledge equivalent, we write $f \equiv g$. For example, functions denoted by $(\cdot \bmod 2)$ and $(\cdot \bmod 4)$ (with postfix notation) are related by $(\cdot \bmod 2) \sqsubseteq (\cdot \bmod 4)$ (knowing the last bit discloses less information than knowing the last two bits). However, $(\cdot > 0)$ is incomparable to either of them.

3 Data Minimisers

We will consider two types of minimiser depending on how the data is collected: *monolithic* and *distributed*. Monolithic collection describes the simple case when personal data is collected from a single location. In the distributed case, inputs from multiple subjects, we assume that we cannot minimise one input using knowledge of the other inputs in the setting of Fig. 2. The aim now is to defined a notion of data minimiser – a program to be deployed at a data source (one per source in the distributed case) – which can remove information from data in a way that (in the best case) the remaining information is enough for the server to compute the intended function p, and (in the best case) nothing more.

Semantics of Minimisation. Let us consider the program in Fig. 3 to discuss the need for a semantic definition of minimisation. In this program, `x2` is syntactically needed to evaluate the condition (ℓ.3). However, this condition always evaluates to `true`. In the same way, `x3` is not semantically needed to compute the value of `y` since it is both added and substracted (ℓ.4). As a consequence, it would be possible to get the same result by rewriting the program with only the input `x1` without modifying its behavior. If `x2` and `x3` are personal data, then the semantic approach is better likely to help us to limit the collection than the syntactic approach. So, the program could be refactored by taking only the input `x1` while retaining the same output behaviour. Though this would work, it requires a change in both the data collected and in the interface of the program.

Instead, we propose to keep the program unchanged and we rely on the idea that the information behind `x2` and `x3` (in this specific example) can be protected by providing *fixed arbitrary* values for them, instead of refactoring the program. This means the program does not need to be modified for the data processor to propose better guarantees. This approach allows a better modularity and a better integration in the software development cycle. To see this, let us consider the program shown in Fig. 1. In this case, any employee earning less than $10000 can disclose any figure between 0 and 9999 and any employee earning at least $10000 can disclose any figure between 10000 and 100000 without affecting the output of the program. Thus a corresponding *data minimiser* could be as shown in Fig. 4, where the representative values are taken to be `0` and `10000`.

The value output by this minimiser, which is the value of `repr_salary`, can then be provided as input to the corresponding program (standing for the variable `salary`). The behaviour of the program will remain the same while the actual salary of the employee is not needlessly passed to the program. This holds even when an employee earns exactly the amount of `repr_salary` since the data processor receiving this value cannot distinguish when it is the case or not.

```
1 input(x1, x2, x3);
2 y := 0;
3 if (x2 == x2;)
4     then (y := (x3 + x1 - x3);)
5 output(y);
```

Fig. 3: Example of a semantically unnecessary syntactic necessity.

Following this approach, we study both monolithic and distributed cases as shown in Fig. 2 where M and $\bigotimes_{i=0}^{n} M_i$ are a monolithic and a distributed minimiser respectively (each M_i is a local minimiser), P is the program, u_i are input values, r_{u_i} are input representatives, and o are outputs.

```
1 input(salary)
2 repr_salary := 0
3 if (10000 <= salary)
4     then (repr_salary := 10000)
5 output(repr_salary)
```

Fig. 4: Minimiser for P_{bl} (see Figure 1).

3.1 Monolithic Case

Let us assume a program P constituting a legitimate processing of personal data for which the data subject had consented to the purpose. We abstract it through a function p which ranges over $\mathcal{I} \to \mathcal{O}$. Since we aim at building *pre-processors* for the program, we consider a minimiser m with type $\mathcal{I} \to \mathcal{I}$.

Definition 3 (Monolithic minimiser). *We say m is a* monolithic minimiser *for p iff (i) $p \circ m = p$ and (ii) $m \circ m = m$.*

Condition (i) states correctness: using the pre-processor does not change the result of the computation. Condition (ii) ensures that m chooses representative inputs in a canonical way, i.e., m removes information from the input in one go.

3.2 Distributed Case

In general a computation over private data may require data collected from several independent sources. We will thus extend our definition to the distributed setting. This is the general case where a program dp is a function of a product of input domains, $dp \in \prod_{i=0}^{n} \mathcal{I}_i \to \mathcal{O}$. The idea for the distributed minimiser will be to have a *local minimiser* $m_i \in \mathcal{I}_i \to \mathcal{I}_i$ for each source $\mathcal{I}_i$, combined into an input processor as $dm = \bigotimes_{i=0}^{n} m_i$, where for $f \in A \to A'$ and $g \in B \to B'$ we have $f \otimes g \in A \times B \to A' \times B'$. This is based on the assumption that each argument of dp is provided by a different data source.

Definition 4 (Distributed minimiser). *We say dm is a* distributed minimiser *for dp iff (i) dm is a monolithic minimiser for dp and (ii) dm has the form* $\bigotimes_{i=0}^{n} dm_i$ *(with* dm_i *being* $\mathcal{I}_i \to \mathcal{I}_i$ *functions — called local minimisers, for* $i \in \{0, \ldots, n\}$*).*

The first condition ensures that *dm* is actually a minimiser while the second one ensures that each input is treated independently. The latter is the key differentiator between monolithic and distributed minimisers. Each function dm_i is only provided with an input from $\mathcal{I}_i$. Intuitively, this means that if two input values at a given position belong to the same equivalence class, then this must hold for all possible values provided for the other positions. For example, if the values 0 and 1 from $\mathcal{I}_0$ belong to the same equivalence class, then each pair of input tuples $\langle 0, x, y\rangle$ and $\langle 1, x, y\rangle$ for all $(x, y) \in \prod_{i=1}^{2} \mathcal{I}_i$ have to belong to the same equivalence class (two by two, meaning these can be different classes for each pair: $\langle 0, 1, 2\rangle$ and $\langle 1, 1, 3\rangle$ may not belong to the same equivalence class, though $\langle 0, 1, 2\rangle$ and $\langle 1, 1, 2\rangle$ have to). This is formally stated in Proposition 1 below, which relies on the definition of the *kernel* of a function, which induces a partition of the input set where every element in the same equivalence class is mapped to the same output.

Definition 5 (Kernel of a function). *If* $f \in \mathcal{I} \to \mathcal{O}$ *then the* kernel of f *is defined as* $\ker(f) = \{(u, v) \mid f(u) = f(v)\}$.

Proposition 1. *If m is a monolithic minimiser for p, then m is a distributed minimiser for p iff for all* $(\boldsymbol{u}, \boldsymbol{v}) \in \ker(m)$*, for all input positions i, and all input vectors* $\boldsymbol{w}$*,* $(\boldsymbol{w}[i \mapsto u_i], \boldsymbol{w}[i \mapsto u_i]) \in \ker(m)$ *where the notation* $\boldsymbol{w}[i \mapsto u_i]$ *denotes a vector like* $\boldsymbol{w}$ *except at position i where it has value* u_i*.*

This proposition gives a data-based characterisation of *data minimisation*, which will be useful when building minimisers in Section 5. Before building minimisers, we explain how to compare them in the next section.

4 Best Minimisers

Now that we defined minimisers as pre-processors modifying the input, we see that there may exist many different minimisers for a given program. Thus we are interested in being able to compare these minimisers. Indeed, since the identity function is a minimiser for all programs p, then it is clear that the simple existence of a minimiser does not guarantee any kind of minimality. One way to compare minimisers is to compare the size of their ranges – a smaller range indicates a greater degree of minimality (cf. Proposition 2 below). A more precise way to compare them is by understanding them in terms of the *lattice of equivalence relations* [25]. The set of equivalence relations on $\mathcal{I}$ forms a complete lattice, with the ordering relation given by set-inclusion of their defining sets of pairs. The identity relation (denoted by $\mathrm{Id}_{\mathcal{I}}$) is the bottom element, and the total relation (denoted by $\mathrm{All}_{\mathcal{I}}$) is the top.

The following proposition provides some properties about the order relation between programs (cf. Definition 2), including its relation with the kernel.

Proposition 2 (Disclosure ordering properties).
(1) $f \sqsubseteq g$ *iff* $\ker(f) \supseteq \ker(g)$ *(2)* $f \circ g \sqsubseteq g$ *(3)* $f \sqsubseteq \langle f, g \rangle$
(4) $\langle f, f \rangle \sqsubseteq f$ *(5)* $f \sqsubseteq g$ *implies* $|\text{range}(f)| \leq |\text{range}(g)|$

where $\langle f, g \rangle$ is the function $x \mapsto (f(x), g(x))$ (see Section 2).

4.1 Monolithic Case

The disclosure ordering allows us to compare the minimisers of a given program, defining a *best* minimiser to be one which discloses no more than any other.

Definition 6 (Best monolithic minimiser). *We say that* m *is a* best monolithic minimiser *for* p *iff (i)* m *is a monolithic minimiser for* p *and (ii)* $m \sqsubseteq n$ *for all minimisers* n *for* p.

In this simple (monolithic) form, minimisation is easily understood as injectivity.

Proposition 3 (Monolithic best minimiser injectivity). *A monolithic minimiser* m *for a program* p *is a* best *one iff* $p|_{\text{range}(m)}$ *is injective (with* $p|_{\text{range}(m)}$ *the restriction of the program* p *over the range of* m*).*

Now we can show that using a minimiser m at the client guarantees that the attacker $\langle p, h \rangle$ learns no more about the input than which can be learned by observing the output of the legitimate use p (the proof follows from the ordering between minimisers and Proposition 2).

Theorem 1. *If* m *is a best monolithic minimiser for* p *then for all hidden uses* h *we have* $\langle p, h \rangle \circ m \equiv p$.

The proof of the following theorem proceeds by building a best monolithic minimiser from kernel of p.

Theorem 2. *For every program* p *there exists a best monolithic minimiser.*

4.2 Distributed Case

As for the monolithic case, we have an ordering between distributed minimisers. We define the notion of a best minimiser for the distributed case as follows ($\sqsubseteq$ is the order between functions, i.e. the inverse of the kernel set inclusion order).

Definition 7 (Best distributed minimiser). *We say* dm *is a* best distributed minimiser *for* dp *iff (i)* dm *is a distributed minimiser for* dp *and (ii)* $dm \sqsubseteq dn$ *for all* dn *distributed minimisers for* dp.

In the following we show that there always exists a best distributed minimiser.

Theorem 3. *For every distributed program* dp *there exists a best distributed minimiser.*

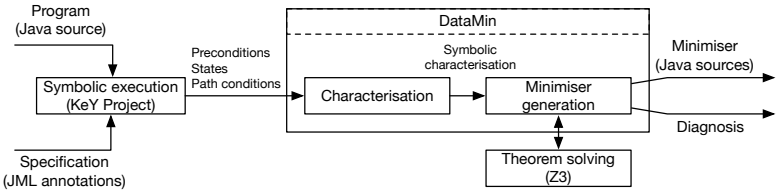

Fig. 5: Toolchain of the (semi-)procedure to generate minimisers.

Note that there is a price to pay here: the best distributed minimiser may reveal more information than the best monolithic minimiser. An example is the *OR* function where the identity function is a best distributed minimiser, but in the monolithic case we can minimise further by mapping $(0, 0)$ to itself, and mapping all other inputs to $(1, 1)$. Similarly to Proposition 1, we give a data-centric characterisation of best distributed minimisers as follows.

Proposition 4 (Data-based best distributed minimisation). *If dm is a* best *distributed minimiser for dp, then for all input positions i, for all v_1 and $v_2 \in \mathcal{I}_i$ such that $v_1 \neq v_2$, there is some $\boldsymbol{u} \in \mathrm{range}(dm)$ such that $dp\,(\boldsymbol{u}[i \mapsto v_1]) \neq dp\,(\boldsymbol{u}[i \mapsto v_2])$.*

5 Building Minimisers

We describe here how data minimisers are built. This (semi-)procedure is not complete for the best minimisers since obtaining a best minimiser is not computable in general. Besides, and more pragmatically, our procedure is built on top of a theorem prover and a symbolic execution engine, and it is thus limited by the power of such tools. Our procedure follows the toolchain depicted in Fig. 5. Before describing the procedure in detail, we briefly recall concepts related to *symbolic execution*.

5.1 Symbolic Execution

This step corresponds to the two leftmost boxes of the toolchain shown in Fig. 5. Symbolic execution has been pioneered in [17] as a way to reason about symbolic states reached by a program execution. A *symbolic execution* of a program is an execution of the program against symbolic inputs. Expressions are symbolically evaluated and assigned to variables. The state of this execution includes the values of the program (modelled as a mapping called *store*), and *path conditions* (boolean expressions ranging over the symbolic inputs, specifying the conditions to be satisfied for a node to be reached). Assignments modify the symbolic value of variables, whereas conditions and loops create branches distinguished by their path conditions. This execution generates a *symbolic execution tree* where each node is associated to a statement and each arc to a transition of the program. A state is associated to each node and the root node corresponds to the input of

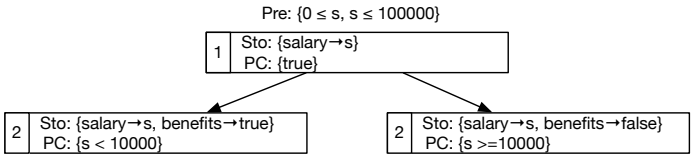

Fig. 6: Symbolic execution tree of P_{bl} (see Figure 1).

the program. Each leaf in the tree corresponds to a completed execution path, and no two leaves have the same path condition.

Let us consider again program P_{bl} from Fig. 1. A representation of the possible paths for P_{bl} is shown in Fig. 6 where the digits 1 and 2 correspond to the lines of the program, Sto is the corresponding store, and PC is the path condition at this point ($\ell.3$ of the program does not have any effect in this representation). The possible execution paths for this program lead to two main outputs, where either `benefits == true` or `benefits == false`. Thus the best minimiser should distinguish the two cases and return a representative value leading to the same output as if the program were to be executed with the "real" value assigned to `salary`. This is the case for the data minimiser described in Fig. 4. Moreover, any change of value for `repr_salary` leads to a change in the `salary` computed by the main program: this minimiser is indeed a best minimiser.

Loops in programs require an invariant. Failing to provide good invariants results in weaker path conditions failing to capture all of what is theoretically knowable about the states of the symbolic execution. To find the best invariant is a difficult problem and still an active field of research [13].

In our approach the program P is symbolically executed along with the assertions coming from the specification S. We assume that we have a global *precondition* for the program P, given in the specification S of the program, denoted $\mathrm{Pre}\langle P, S\rangle$. In our example, the conditions $0 \leq s$ and $s \leq 100000$ are part of the preconditions as shown in Fig. 6. This produces a symbolic execution tree equipped with the corresponding path conditions PC and Sto attached to its nodes. In what follows we define a symbolic characterisation of the program P under specification S capturing the conditions at the end of the execution.

Definition 8 (Symbolic characterisation of a program). *We say Γ is a* symbolic characterisation *of a program P under specification S, written $\Gamma_{\langle P,S\rangle}$, iff Γ collects the preconditions, stores, and path conditions to be satisfied for each possible output of P: $\Gamma_{\langle P,S\rangle} \triangleq \mathrm{Pre}\langle P, S\rangle \wedge \Big(\bigvee_{l \in \mathrm{Leaves}\langle P,S\rangle} (\mathrm{PC}(l) \wedge \mathrm{Sto}(l))\Big)$, where $\mathrm{Leaves}\langle P, S\rangle$ returns the leaves of the symbolic execution tree of P under specification S, and $\mathrm{PC}(\cdot)$ and $\mathrm{Sto}(\cdot)$ return the path condition and the state associated to a leaf, respectively.*

For the example in Fig. 1, the (simplified) symbolic characterisation is: $\Gamma_{\langle P_{bl},S_{bl}\rangle} = (0 \leq s \wedge s \leq 100000) \ \wedge \ ((s < 10000 \wedge \mathit{salary} = s \wedge \mathit{benefits} = \mathit{true}) \vee (s \geq 10000 \wedge \mathit{salary} = s \wedge \mathit{benefits} = \mathit{false}))$.

```
1: min = {}                                                    ▷ initialise minimiser
2: decl logical_variable i'                                    ▷ declare a new logical variable
3: for all i ∈ Inputs do                                       ▷ iterate over the inputs
4:     γ = Γ                          ▷ copy the program symbolic characterisation to γ
5:     min[i] = {}                                             ▷ initialise minimiser for input i
6:     while γ.check() do             ▷ call solver to loop as long as γ is satisfiable
7:         model = γ.model()          ▷ call solver to get a valid assignment for γ
8:         formula = ((i' == model[i]) ∧ Γ ∧ Γ[i'/i])         ▷ build distributed min. formula
9:         wp = formula.quantif_elim()    ▷ call solver to eliminate quantifiers
10:        min[i]+= (wp, model[i])  ▷ add weakest precond. with representative to min.
11:        γ = γ ∧ ¬wp  ▷ conjunct negation of the weakest precond. to γ to limit the loop
12:    end while
13: end for
```

Fig. 7: Distributed data minimiser generation (excerpt).

The symbolic characterisation induces an equivalence class giving a partition of the state space according to the different possible outputs. A solver uses this to get a representative for each class, which is the next step in our procedure.

5.2 Static Generation of Minimisers

We show here how best minimisers are generated from the symbolic execution tree. The monolithic case is a particular case of the distributed one. The construction spans over the *DataMin* box shown Fig. 5.

The input domain of the semantic function dp having $n+1$ inputs is $\prod_{i=0}^{n} \mathcal{I}_i$ and the corresponding input variables of dp are denoted as x_i. A local function dm_i is generated for each input $x_i \in \text{Input}(DP)$. We are still interested in the different possible outputs of dp but we cannot directly use its kernel as in the monolithic case since this would require all the minimisers to have access to the data at other data source points. Instead of this, each data source point assumes that the data to be disclosed by other points could take any possible value in their domain. Thus we need to compute, for each input variable x_i, the equivalence classes over its domain such that the value o of the output of the program remains the same for all possible assignments to the other inputs variables x_j for all $j \in (\{0, \ldots, n\} \setminus \{i\})$.

The algorithm of Fig. 7 shows how the distribution of the inputs is taken into account. Here, *min* (ℓ.1) stands for the minimiser being built (as a map containing the local minimiser *min*[i] (ℓ.5) for each input i (ℓ.3)), *Inputs* (ℓ.3) denotes the inputs of the program, and Γ (ℓ.4) denotes the formula of the symbolic characterisation of the program. The notation $\phi[y/x]$ denotes the formula ϕ in which y replaces x and == the logical equality (ℓ.8).

Primitives depending on a theorem prover are called at three locations in the algorithm. The first one, check() (ℓ.6) checks whether or not a logical formula is satisfiable. Then, model() (ℓ.7) applied to a formula returns a satisfying valuation. These two primitives are linked as it is only possible to find such a valuation for satisfiable formula. Finally, quantif_elim() (ℓ.9) is a procedure to eliminate (both universal and existential) quantifiers in a formula, thus simplifying a formula by removing the corresponding symbols.

After initialising *min*, holding the minimisers as a map from inputs i to tuples (*weakest precondition*, *representative*) (ℓ.1), a new logical variable i' is declared (ℓ.2). At this point, all the inputs (ℓ.3) and the output o of the program already exist as logical variables (implicitly introduced with Γ which is an input of this algorithm). The new logical variable i' is used to control variations on one input during the procedure (ℓ.8). Then, the algorithm iterates over all the inputs (ℓ.3). The symbolic characterisation Γ is assigned to the variable γ (ℓ.4) (which will be reinitialised at each iteration of the loop of ℓ.3). The original Γ will be used to build formulas to be solved while the fresher γ will be used to control the loop (ℓ.6). The minimiser *min* is then initialised for the current input i (ℓ.5). The algorithm loops over all equivalence classes for the current input i. This is ensured by (i) representing each equivalence class by its weakest precondition *wp* (ℓ.7-10), (ii) conjuncting the negation of the weakest precondition *wp* found to the symbolic characterisation γ (ℓ.10), and (iii) looping as long as there is another equivalence class to be covered by checking the satisfiability of γ conjuncted with the negation of all previous conditions (ℓ.6).

We now explain in more detail how the weakest preconditions are found (ℓ.7-10). A satisfying valuation of the characterisation γ is requested from the solver and assigned to the variable *model* (ℓ.7). The valuation of variable x can be called by $model[x]$. A formula (assigned to the variable *formula*) is then built in two steps. First, we conjunct Γ, $\Gamma[i'/i]$, and the formula $i' == model[i]$ (ℓ.8). This fixes $\Gamma[i'/i]$ to the case where i' is equal to the satisfying valuation found previously (ℓ.7). Once the formula has been built, the quantifiers are eliminated by calling a solver (ℓ.9). This gives the weakest precondition corresponding to an equivalence class of the inputs (the one corresponding to the value of $model[i]$). This equivalence class *wp* is then added to the minimiser $min[i]$ along with its representative $model[i]$ (ℓ.10) before being excluded from the control γ (ℓ.11) before a new iteration, until exhaustion of the equivalence classes (ℓ.6).

This algorithm builds a map function *min* which is used to generate the code for the atomic minimisers in a post-processing step (not shown here).

Theorem 4 (Soundness). *The algorithm of Fig. 7 builds a best distributed minimiser dm for program dp.*

Soundness of this algorithm relies on Proposition 4, on the fact that the representative assigned to each equivalence class is fixed, on the proof of existence from Theorem 3, and on the soundness of the external procedures called.

Theorem 5 (Termination). *The algorithm of Fig. 7 terminates when the number of inputs and the number of equivalence classes on these inputs are finite, and the calls to the external solver terminate.*

This is proven by showing that all the loops in the algorithm iterate over sets built from inputs and equivalence classes. The set of inputs is not modified in the loops while the condition of (ℓ.6) strictly decreases in terms of the number of times it is satisfiable. However, it terminates only if there is a finite number of equivalence classes for the current input. Since we depend on external procedures, termination of our algorithm also depends on the termination of such procedures.

5.3 *DataMin* Implementation

The (semi-)procedure described in this section has been implemented in Python as a proof of concept named *DataMin*.[4] We rely on the Symbolic Execution Engine of the *KeY* Project [16]. This symbolic executor is run against a program P written in Java for which a minimiser should be generated. *DataMin* generates the symbolic characterisation $\Gamma_{\langle P,S\rangle}$ and builds the partitioning $k_{\langle P,S\rangle}$ and the sectioning $r_{\langle P,S\rangle}$ functions. The theorem prover *Z3* [24] is called through its API to solve constrains as needed. We currently support only a limited range of data structures but could be extended thanks to the many theories on which *Z3* relies.

Finally, *DataMin* generates the minimiser as a set of Java files to be run by the data source points before disclosing the data. These files are directly compilable and ready to be exported as Java libraries to ease their use. This whole process runs in reasonable time for the examples provided in the archive (less than a second, given that Python is run in its interpreted mode). For the first example (monolithic with a loop) the solver used (*Z3*) was not able to totally eliminate quantifiers. We thus manually eliminated quantifiers by using the *Redlog* system [11]. This limitation comes from the external tools and not from the procedure proposed. The second example (distributed) does not suffer from this limitation and shows how multiple atomic minimisers are generated.

6 Final Discussion

We provided a formal definition of *data minimisation* in terms of strong dependency and derived concepts, and introduced the concept of a *data minimiser* defined as a pre-processor to the data processor. We considered both the monolithic and distributed cases. Finally, we provided a proof-of-concept implementation to obtain data minimisers for a given program. Our approach is semantics-based, so finding a distributed minimiser is undecidable in general.

Formal and rigorous approaches to privacy have been advocated for some time [29], but the data minimisation principle has not been precisely defined in the past, as stated in [15]. A related work is the notion of *minimal exposure* [3], which consists in performing a preprocessing of information on the client's side

[4] `http://www.cse.chalmers.se/research/databin/files/datamin.zip`.

to give only the data needed to benefit from a service. In this setting, if the requested service can be modeled as $a \vee b$, then if a is true then only a will be disclosed: they reduce the *number* of inputs provided, but do not reason on the domain of the inputs as we do.

Minimality is closely related to information flow, and we have used several ideas from that area [10, 18, 27]. A semantic notion of dependency has been introduced in [22], where it is used to characterise the variables in an expression that are semantically relevant for the evaluation of the expression. Our notion is related to this but much more fine-grained. A notable difference in the formalisation of minimality compared to usual definitions of information flow is that the former is a necessary and sufficient condition on information flow, whereas most security formulations of the latter are sufficient conditions only (e.g., noninterference: public inputs are sufficient to compute public outputs). An exception is Chong's *required information release* [9] which provides both upper and lower bounds on information flows. For this reason many static analysis techniques for security properties (e.g. type systems) are not easy to use for minimality — an over-approximation to the information flows is not sound for minimality. The necessary and sufficient conditions embodied in minimisers appear to be closely related to the notion of *completeness* in abstract interpretation [14], where a minimiser m plays the role of a *complete abstraction*. Some work on quantitative information flow aiming at automated discovery of leaks also rely on analysis of equivalence classes [6, 20]. We could use several of the ideas in [6] to improve our implementation.

Equivalence partitioning by using symbolic execution was first introduced for test case generation in [26], and later used by the *KeY* theorem prover [2]. Symbolic execution has limitations, especially when it comes to handling loops. Though being a main concern in theory and for some applications, *while* loops do not seem to be as widespread as *for* loops in practice. For instance, Malacaria et al. have been able to perform a symbolic execution-based verification of non-interference security properties from the C source of the real world *OpenSSL* library [21]. Different other techniques relying on symbolic execution and SAT solvers are presented in [30] to form *nsqflow*, a tool dedicated at measuring quantitative information flow for large programs. They also target realistic programs and thus show the feasibility to decide about non-interference properties in programs.

Acknowledgements. This research has been supported by the Swedish funding agency SSF under the grant *DataBIn: Data Driven Secure Business Intelligence*.

References

1. Antignac, T., Sands, D., Schneider, G.: Data Minimisation: a Language-Based Approach (Long Version). eprint arXiv:1611.05642 (2016).
2. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Soft. & Sys. Model. 4, 32–54 (2005).

3. Anciaux, N., Nguyen, B., Vazirgiannis, M.: Limiting data collection in application forms: A real-case application of a founding privacy principle. In: PST 2012.
4. Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: Proc. IEEE Symp. on Security and Privacy 2007.
5. Askarov, A., Chong, S.: Learning is change in knowledge: Knowledge-based security for dynamic policies, CSF 2012.
6. Backes, M., Kopf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: IEEE Security and Privacy, 2009.
7. Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. PLAS 2011.
8. Bussard, L.: Encyclopedia of Cryptography and Security, chap. Secondary Use Regulations, Spring 2011
9. Chong, S.: Required information release. CSF 2010.
10. Cohen, E.: Information transmission in computational systems. SIGOPS Oper. Syst. Rev. 11(5), 1977.
11. Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. SIGSAM Bull. 31(2), 2–9, 1997.
12. European Commission: General Data Protection Regulation. Codecision legislative procedure for a regulation 2012/0011.
13. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. POPL, 2002.
14. Giacobazzi, R., Ranzato, F.: Completeness in abstract interpretation: A domain perspective. In: AMAST, LNCS 1349 Springer (1997).
15. Gurses, S., Troncoso, C., Diaz, C.: Engineering privacy by design reloaded. Amsterdam Privacy Conference (2015).
16. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (sed). Runtime Verification, Springer 2014.
17. King, J.C.: Symbolic execution and program testing. CACM 19(7), 385–394 1976).
18. Landauer, J., Redmond, T.: A lattice of information. CSFW, 1993.
19. Leavens, G.T., Baker, A.L., Ruby, C.: Jml: a java modeling language. In: Formal Underpinnings of Java Workshop, 1998.
20. Lowe, G.: Quantifying information flow. CSFW, 2002.
21. Malacaria, P., Tautchning, M., DiStefano, D.: Information Leakage Analysis of Complex C Code and Its application to OpenSSL, LNCS 9952, Springer 2016.
22. Mastroeni, I., Zanardini, D.: Data dependencies and program slicing: From syntax to abstract semantics. PEPM 2008.
23. Mitchell, R., McKim, J., Meyer, B.: Design by Contract, by Example. Addison Wesley, 2002.
24. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. TACAS, LNCS 4963, Springer 2008.
25. Oystein, O.: Theory of equivalence relations. Duke Mathematical Journal 9, 573–627 (1942).
26. Richardson, D.J., Clarke, L.A.: A partition analysis method to increase program reliability. ICSE 1981.
27. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. Higher-Order and Symbolic Computation 14(1), 2001.
28. Smith, G.: On the foundations of quantitative information flow. FOSSACS, LNCS 5504, 2009.
29. Tschantz, M.C., Wing, J.M. Formal Methods for Privacy, in FM 2009.
30. Val, C.G., Enescu, M.A., Bayless, S., Aiello, W., Hu, A.J.: Precisely measuring quantitative information flow: 10k lines of code and beyond. EuroS&P 2016.