

# Multi-run security

Arnar Birgisson and Andrei Sabelfeld

Chalmers University of Technology, 412 96 Gothenburg, Sweden

**Abstract.** This paper explores information-flow control for batch-job programs that are allowed to be re-run with new input provided by the attacker. We argue that directly adapting two major security definitions for batch-job programs, termination-sensitive and termination-insensitive noninterference, to multi-run execution would result in extremes. While the former readily scales up to multiple runs, its enforcement is typically over-restrictive. The latter suffers from insecurity: secrets can be leaked in their entirety by multiple runs of programs that are secure according to batch-job termination-insensitive noninterference. Seeking to avoid the extremes, we present a framework for specifying and enforcing multi-run security in an imperative language. The policy framework is based on tracking the attacker’s knowledge about secrets obtained by multiple program runs. Inspired by previous work on robustness, the key ingredient of our type-based enforcement for multi-run security is preventing the dangerous combination of attacker-controlled data and secret data from affecting program termination.

## 1 Introduction

Imagine a scenario of a web service with a medical database at the back-end. Analysts are allowed to access the database through a web interface. The goal is to allow deriving interesting statistics (say, by age groups or by larger residential areas) but disallow leaking sensitive information about individuals. In this scenario, the server-side program that accommodates queries has two inputs: one is the database itself, which contains sensitive data and which is not controlled by the attacker, and the other one is a public query that originates from a possibly malicious analyst. For the program to function, it must have access to the entire database. At the same time, it must not reveal secret data contained in the database. Hence, we are interested in securing *information flow* from secret inputs to public outputs. This problem arises both when the code is written by non-malicious developers, in which case we want prevent accidental leaks, and when the code is supplied by untrusted third parties, when we want to prevent malicious leaks. Settling for the worst case, we do not appeal to trust assumptions.

Language-based information-flow security [31] is focused on providing strong security guarantees for underlying programs. In the context of confidentiality, it is intended to prevent information flow from secret inputs to public outputs. The dominating baseline security policy is *noninterference* [14, 19] that requires that a variation of secret input does not result in a variation of public outputs.

However, the state of the art in the area consists of two extremes. One extreme is batch-job program models, where programs are run only once and where the initial memory is the only input and the final memory is the only output. A large body of research on language-based information-flow security is limited to batch-job models. In a

language-based setting, noninterference has been largely considered for batch-job models [40, 31]. Major efforts on information flow in functional [28], object-oriented [43, 8, 20], concurrent [37, 42, 30], and other languages [31] assume a batch-job model.

While securing batch-job programs without being over-restrictive is feasible, the assumption that programs are run only once is often too strong. The other extreme is fully interactive programs with channels for input/output communication. While this model is more powerful, securing interactive programs is notoriously hard: intermediate observations can be exploited to leak information [2].

This paper explores middle ground between the extremes: batch-job programs that are allowed to be re-run with new input provided by the attacker. We believe this model captures many practical scenarios such as the medical database above. Our attacker model allows issuing queries to the database as described by a batch-job program whose secret input is the database and public input is the attacker-controlled part of the query. The goal is to prevent the attacker from learning sensitive information by re-running the program with modified public parameters and observing the public outcome.

Leaks via termination behavior of programs turn out to be the bottleneck for generalizing batch-job style security to multiple runs. We argue that directly adapting two major security definitions for batch-job programs, termination-sensitive and termination-insensitive noninterference, to multi-run execution would result in further extremes. The former, *termination-sensitive noninterference* [39, 31], readily scales up to multiple runs. This definition demands that the public outcome and termination behavior of the underlying program do not depend on secret data. No run leaks any information about secrets, and so we can safely re-run programs. Thus, batch-job termination-sensitive security implies multiple-run security. However, enforcing termination-sensitive noninterference withing being overly restrictive is far from trivial. Typically, enforcement mechanisms (e.g., [39]) place Draconian restrictions whenever abnormal termination is possible in sensitive context. For example, no sensitive data is allowed in loop guards.

The latter, *termination-insensitive noninterference* [40, 31], where secrets are allowed to affect termination behavior, suffers from insecurities in the multi-run case. Let us illustrate the problem with examples. The program

```
while  $h$  do skip (1)
```

where  $h$  contains a secret, is deemed secure. Termination-insensitive noninterference quantifies over all possible input memories that agree on the public part and makes sure that terminating runs agree on the public part of the final memories. The termination behavior is not considered to have a significant effect, even though the termination depends on secret data. Although the condition quantifies over possible runs, its guarantees are only about differences between two single runs. The implicit assumption is that the program is run only once. A common argument is that if a batch-job program that satisfies termination-insensitive noninterference is run only once, then it leaks at most one bit [2].

With the same rationale, flavors of this program are also accepted by mainstream information-flow security tools Jif [26], FlowCaml [35], and the SPARK Examiner [9, 12] for Java, Caml, and Ada, respectively.

Similarly, the program

```
while  $h = l$  do skip (2)
```

where  $h$  contains a secret and  $l$  is an attacker-controlled public variable, is also considered secure.

However, the single-run assumption is in many cases inadequate. As in the database scenario above, attackers are often capable of re-running the program. Further, in a smartcard setting, the attacker may try to leak the secret key by multiple attempts of feeding public inputs and observe the properties of public outputs. A web attacker can initiate multiple runs of a server-side computation that involves secrets by providing a request with public input. Similarly, the attacker can initiate multi-run computation on the client side of an honest user by providing scripts that keep re-running after recovering from divergence (rather straightforward to accomplish with the modern browsers' interpretation of JavaScript). A recent exploit of ASP.NET analyzes the difference between error messages of multiple requests to collect information for a padding oracle attack [29].

This ability does not make a difference for program 1 (given the value of the secret is unchanged between the runs), but it is fatal for program 2: the attacker can learn the entire secret by brute-force guessing the value of  $h$  with different choices for  $l$ . Multi-run leaks are particularly devastating for single-run secure programs like:

```
while  $h \&\& l$  do skip (3)
```

where  $\&\&$  is the bitwise “and” operation. By walking through the bits of  $h$  in subsequent runs, the attacker can learn the entire value in linear time (of the bit-size of the secret) bit-by-bit. Thus, secrets can be leaked in their entirety by multiple runs of programs that are single-run secure. A quick experiment with a Jif-certified program that contains a termination leak of this kind shows that it is straightforward to leak one secret bit per second even on a modest modern desktop machine (tested with Jif 3.0). This implies that a credit card number can be leaked within a minute. The Jif program and a simple Python script that exploits its termination leak are shown in Appendix B.

Seeking to avoid the extremes, we present a framework for specifying and enforcing multi-run security. For specification, we are inspired on knowledge-based attacker models [17, 4]. The policy framework is based on tracking the attacker's knowledge about secrets obtained by multiple program runs. The framework supports possibilities for intended information release (illustrated by examples below). Further, it connects to quantitative security, where we reason about how many bits of information can be leaked by multiple program runs.

For enforcement, we are inspired by previous work on robustness [41, 25, 3]. The key ingredient of our type-based enforcement for multi-run security is preventing the dangerous combination of attacker-controlled data and secret data from affecting program termination. It is particularly gratifying that we can draw on the type system for robustness for enforcing a policy that it has not been designed for. This connection leads us to clean enforcement, providing a simple solution to a nontrivial problem of multi-run security.

For information-flow tracking, we deploy data labels that combine confidentiality and integrity information. Confidentiality distinguishes secret information from public by *high* and *low* confidentiality labels. Integrity distinguishes untrusted information from trusted by *low* and *high* integrity labels. For confidentiality the use of high information is more restrictive: secrets may not leak to public; and dually for integrity use of low is restricted: untrusted data may not affect trusted. Typically, lattices [16] are used to reason about more complex structures than low/high for confidentiality and integrity. Of particular interest of us are product lattices that combine confidentiality and integrity labels. In the example of a product lattice that combines two low/high lattices, the top element is high confidentiality and low integrity. Data at this level is most restrictive to use. The bottom element is low confidentiality and high integrity, which may arbitrarily affect data at other levels. Integrity plays a key role for the enforcement: the enforcement ensures that combinations of high-confidentiality (secret) and low-integrity (attacked-controlled) data do not affect the termination behavior.

As foreshadowed above, we extend our approach to specify and track intentional information release (or *declassification*). The extended enforcement guarantees that the program does not release more information than described by *escape-hatch* [32] expressions. The purpose of escape hatches is to describe what is allowed to be released. The job of the underlying security condition is to ensure that *nothing else* about secret data may be learned by the attacker. For example, program

$$l := h \% 4 \tag{4}$$

releases two least-significant bits about the secret variable  $h$ . When this is desired, it is expressed in our framework by the escape-hatch expression  $h \% 4$ . The type system accommodates intentional release by labeling escape-hatch expressions as low confidentiality and high integrity. Hence, the program above is accepted while, for example, program

$$l := h \% 6 \tag{5}$$

is rejected because the type system detects a mismatch with the escape hatch  $h \% 4$ .

Next, assuming the same escape-hatch policy  $h \% 4$ , consider the following program:

$$\text{while } h \% 4 + l \text{ do skip} \tag{6}$$

This program may also release two least-significant bits about  $h$ . Indeed, the attacker may experiment by supplying inputs  $-2$ ,  $-1$ , and  $0$  for  $l$  and observing whether the program diverges. Our type system rightfully accepts this program because the loop guard  $h \% 4 + l$  has low integrity and low confidentiality, inheriting its restrictions from variable  $l$  (recall that  $h \% 4$  is labeled as low confidentiality and high integrity, which is least restrictive).

A final example illustrates how intended declassification is distinguished from unintended. Assuming the escape-hatch policy  $h$ , consider the program

$$h := h'; l := h \tag{7}$$

that attempts to leak the initial value of  $h'$  by *laundering* its value through the declassified (syntactic) variable  $h$ . This program is rejected because the enforcement mechanism detects that a variable involved in declassification has been modified.

## 2 Security condition

This section presents some key definitions, in particular the definition of when we consider programs multi-run secure. Command  $c$  represents a deterministic program in the rest of the paper. As before,  $h$  and  $l$  represent secret (*high*) and public (*low*) variables. Without loss of generality, we treat a program as a function of two inputs (secret and public) coming from some finite domain  $D$ , to the set  $D \cup \{\perp\}$ . The result  $c(h, l)$  expresses the observed low output of the program, with the special value  $\perp$  representing nontermination.

**Definition 1 (Single-run knowledge).** *Let  $c$  be a program taking two inputs, a fixed secret one  $v_h$  and a (non-fixed) public one  $v_l$  each from some domain  $D$ , and yielding a public output  $c(v_h, v_l) \in D \cup \{\perp\}$ .*

*An attacker (with full knowledge of  $c$  itself) is allowed to execute  $c$ , providing the public input  $v_l$  and observing only the public output  $c(v_h, v_l)$ . The attacker's knowledge of the (fixed) secret input is then represented by the set of values that would lead to the observed outcome. This set is written as:*

$$k_{v_h}(c, v_l) = \{x \in D \mid c(x, v_l) = c(v_h, v_l)\}$$

Programs with more than two inputs are modeled by collecting all secret and public inputs into two separate tuples, and similar for programs which have more than one output.

Note that by allowing  $c(v_h, v_l)$  to take the special value  $\perp$  (meaning that  $c$  has an infinite derivation for those inputs), we make nontermination observable. This is important because in reality, nontermination can for example be (approximately) inferred from programs that time out.

Assume an attacker has some previous knowledge of  $h$ , represented by the set  $K_0 \subseteq D$ . Then the attacker is potentially able to increase that knowledge (which corresponds to shrinking the set of possibilities) by running the program. The attacker's new knowledge will be the single-run knowledge intersected with the previous knowledge. Repeating this process (possibly with different low inputs) results in a sequence of increasing knowledge (decreasing sets of possibilities). For an attacker with no initial knowledge we can simply start with  $K_0 = D$ . Obviously, a program may potentially leak more if the attacker gets the chance to invoke it multiple times and has control over some of the input.

The maximum knowledge attainable by the attacker is the result of the above process repeated for every possible low input. Note that this models a powerful attacker, as the number of possibilities is exponential in the bit-size of the input. This maximum knowledge, or *multi-run knowledge* is now defined as follows.

**Definition 2 (Multi-run knowledge).** *Let  $c, v_h, v_l$ , and  $D$  be as in Definition 1. The attacker's knowledge about  $v_h$  produced by multiple runs of program  $c$  is defined as:*

$$K_{v_h}(c) = \bigcap_{v_l \in D} k_{v_h}(c, v_l)$$

To highlight the contrast between multi-run knowledge and single-run knowledge captured by the definitions, we come back to the examples from Section 1. Assume  $D = \{0, \dots, 255\}$ . Recall program 1:

while  $h$  do skip

The single-run knowledge  $k_{v_h}(c, v_l)$  for this program is  $\{0\}$ , when  $c(v_h, v_l) = v_l$ , and  $\{1, \dots, 255\}$ , when  $c(v_h, v_l) = \perp$ . The multi-run knowledge  $K_{v_h}(c)$  is  $\{0\}$ , when  $v_h = 0$ , and  $\{1, \dots, 255\}$ , when  $v_h \neq 0$ , which directly corresponds to the two cases for the single-run knowledge. Recall now program 2:

while  $h = l$  do skip

The single-run knowledge  $k_{v_h}(c, v_l)$  for this program is  $\{0, \dots, v_l - 1, v_l + 1, \dots, 255\}$ , when  $c(v_h, v_l) = v_l$ , and  $\{v_l\}$ , when  $c(v_h, v_l) = \perp$ . However, the multi-run knowledge  $K_{v_h}(c)$  is simply  $\{v_h\}$ , which corresponds to leaking all of  $v_h$  into variable  $l$ . The intersection in the definition of multi-run knowledge corresponds to traversing all possible low inputs in the attempt to match them to  $v_h$ , which is a worst-case model for multi-run attackers.

Now that we have definitions of attacker knowledge obtained after running the program, we wish to express a policy which sets limits on this knowledge. A *knowledge policy* states a lower bound on the attacker's uncertainty by partitioning the input domain into classes. Each class lists values that must remain indistinguishable to the attacker. In other words, the attacker may identify from which class the secret input comes, but any more precision is disallowed. This view corresponds to *partial release* [14, 33] of information. This leads to the following definition.

**Definition 3 (Knowledge-policy).** A knowledge policy  $P$  for an input with domain  $D$  is a partition of  $D$  into classes  $P_i$ :

$$P = \{P_1, \dots, P_n\} \quad P_i \subseteq D \quad i \neq j \implies P_i \cap P_j = \emptyset \quad P_1 \cup \dots \cup P_n = D$$

For a value  $v \in D$  we write  $\llbracket v \rrbracket_P$  to represent the class of  $P$  to which  $v$  belongs.

Note that as a partition of  $D$ , a policy represents an equivalence relation on values. Two values are equivalent if they come from the same class. This equivalence relation is often referred to as an *indistinguishability relation* [14, 33].

We illustrate the definition with simple examples. A policy that allows the attacker no knowledge is simply  $P = \{D\}$ . A policy that allows full knowledge is  $\{\{x\} \mid x \in D\}$ . If  $D$  is the set of unsigned 8-bit integers, then a policy that allows the attacker to know the parity of the secret is:

$$\{\{0, 2, \dots, 254\}, \{1, 3, \dots, 255\}\}.$$

We are now ready to state the formal definition of *multi-run secure programs*.

**Definition 4 (Multi-run security).** Let  $c$  be a program that takes a secret input  $v_h$  and an arbitrary public, attacker-controlled input.  $c$  is multi-run secure (or simply secure) with respect to a knowledge policy  $P$  if and only if  $\llbracket v_h \rrbracket_P \subseteq K_{v_h}(c)$ .

Observe that multi-run security with policy  $\{D\}$  corresponds to *termination-sensitive noninterference* [39, 31] for single runs, which prevents the termination behavior of the program (as well as its public output) from being affected by secrets.

Recall programs 4 and 5 from Section 1. Program 4 ( $l := h\%4$ ) is secure for all high input with respect to the policy  $P = \{\{0, 4, \dots\}, \{1, 5, \dots\}\}, \{2, 6, \dots\}, \{3, 7, \dots\}$ . Indeed, the multi-run knowledge from running the program has to be one of the four sets in the policy because the attacker only learns the two least-significant bits.

On the other hand, program 5 ( $l := h\%6$ ) is insecure for all high input according to the policy  $P$ . To illustrate this, take  $v_h = 0$ . The multi-run knowledge  $K_0(c)$  is  $\{0, 6, \dots\}$ , while  $\llbracket 0 \rrbracket_P = \{0, 4, \dots\}$  which is clearly not contained in the knowledge.

As we are interested in an enforcement mechanism that allows limited leaks through the termination channel, Definition 4 will serve as the basis for a relaxed definition which we apply to the enforcement mechanism presented in Section 3. This relaxation draws on ideas from quantitative security. Smith [36] defines the notion of *vulnerability*  $V(X)$ , which is the worst-case probability of guessing the value of secret  $X$  by an adversary in one try. The measure of information quantity is then defined as  $-\log V(X)$ . Based on the intuition “information leaked = initial uncertainty - remaining uncertainty”, Smith defines *information leakage* and shows that for deterministic programs and uniformly distributed secrets it amounts to  $\log |S|$ , where  $|S|$  is the size of the set of possible public outputs given the public input is fixed. The intuition is that the more different observations the attacker can observe, the more secret information about might leaked through them. In the multi-run case, the size of the set of possible outputs translates to the number of indistinguishability classes for the high input, which, in effect, is the number of different values  $K_{v_h}(c)$  can take when  $v_h$  varies. This is in line with Lowe [23], who measures the number of secret behaviors distinguished by an attacker in a nondeterministic setting. This motivation brings us to the following definition:

**Definition 5 (*k*-bit security).** *Let  $c$  be a program that takes a uniformly distributed secret input  $v_h$  and an arbitrary public, attacker-controlled input  $v_l$ .  $c$  is  $k$ -bit secure if  $k = \log n$  and  $K_{v_h}(c)$  takes at most  $n$  distinct values as  $v_h$  varies.*

For example, program 1 is 1-bit secure because there are only two possibilities for  $K_{v_h}(c)$  as  $v_h$  varies. On the other hand, program 2 is  $k$ -secure, where  $k$  is the bit size of  $h$  because  $K_{v_h}(c)$  ranges over all possible singleton sets as  $v_h$  varies.

1-bit security is a particularly interesting case. Intuitively it means that an attacker can at most infer that  $v_h$  is in some set  $A$  or that it is in  $A$ 's complement. In an extreme case, either set might contain only one element, meaning the attacker would know the exact value of that particular  $v_h$ , but since there are only two possible “knowledges” this is equivalent to the attacker being allowed only one boolean test on the secret.

Ultimately we will prove that our enforcement mechanism is multi-run secure with respect to a policy, with the relaxation that 1-bit leaks are allowed. For simplicity we combine Definition 4 and the 1-bit version of Definition 5 as follows.

**Definition 6 (1-bit security w.r.t. a policy).** *Assume  $c, v_h, v_l$  are as in Definition 5, and  $P$  is a knowledge policy. We say that  $c$  is 1-bit secure with respect to  $P$  if and only*

$$\begin{aligned}
n &\in D, \quad x \in \text{Vars}, \quad \text{op} \in \{+, -, \dots\} \\
e &::= n \mid x \mid e \text{ op } e \\
c &::= \text{skip} \mid x := e \mid c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c
\end{aligned}$$

**Fig. 1.** Syntax

if for each class  $P_i \in P$ ,  $K_{v_h}(c)$  takes at most two distinct values (knowledges)  $K_1, K_2$  as  $v_h$  varies within  $P_i$ , and furthermore  $P_i \subseteq K_1 \cup K_2$ .

In other words,  $K_{v_h}(c)$  can vary arbitrarily for  $v_h$  from different indistinguishability classes, but within each class we only allow for revealing one additional bit of information. The last part ensures that an attacker cannot otherwise exclude any values from the policy class of the secret, any value considered impossible in one knowledge must be considered possible according to the other knowledge.

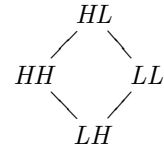
### 3 Enforcement

We illustrate our approach to enforcement for an imperative language. To keep the exposition clear, we have deliberately chosen a simple language, but the ideas here scale to more complex languages. Figure 1 shows the syntax of the language. Expressions take literals from a finite domain  $D$  (e.g., 32-bit integers) and variables from a set  $\text{Vars}$ . We present a type system for this language such that typable programs are robust against multi-run attacks that try to magnify single-run termination leaks into leaking more than one bit. The type system represents a static analysis, conveniently referring to security labels for variables and expressions as *security types*.

We will continue to treat programs as functions  $D \times D \rightarrow D \cup \{\perp\}$ , and in concrete examples the inputs will be represented by the variables  $h$  and  $l$ . The final value of  $l$  will be the output for terminating programs.

The important feature of this type system is that it does not allow looping on expressions that both depend on secrets *and* attacker input. Thus we need to consider both the confidentiality and integrity levels of expressions at the same time. To achieve this we label variables with labels from the following product lattice  $\mathcal{L}$  that combines confidentiality and integrity.

Here a label lists first the confidentiality level and then the integrity level. For example, the attacker provided input  $l$  has level  $LL$  (low confidentiality, low integrity) since it is both known and controlled by the attacker, and the secret input  $h$  has level  $HH$  since it is neither. An expression combining a secret with untrusted input will be assigned level  $HL$  (high confidentiality, low integrity). We



use the standard symbols  $\sqsubseteq, \sqcup$ , etc. for lattice operators. This lattice has been used for enforcing *robust declassification* [41, 25, 3], which demands that the attacker may not affect what is released by programs by ensuring that only high-integrity data can be declassified, and only in a high-integrity context. The work on robust declassification is a direct inspiration for our treatment of the termination channel in multi-run security.

$$\begin{array}{c}
\text{SKIP} \frac{}{pc \vdash \text{skip}} \quad \text{ASSIGN} \frac{lev(e) \sqcup pc \sqsubseteq lev(x)}{pc \vdash x := e} \quad \text{SEQ} \frac{pc \vdash c_1 \quad pc \vdash c_2}{pc \vdash c_1; c_2} \\
\text{IF} \frac{pc \sqcup lev(e) \vdash c_1 \quad pc \sqcup lev(e) \vdash c_2}{pc \vdash \text{if } e \text{ then } c_1 \text{ else } c_2} \quad \text{WHILE} \frac{pc \sqcup lev(e) \vdash c \quad pc \sqcup lev(e) \neq HL}{pc \vdash \text{while } e \text{ do } c}
\end{array}$$

**Fig. 2.** Typing rules

However, as we explain in Section 4, the policy that robust declassification enforces is rather different from our security model. Our observation that connects robust declassification with multi-run security enables us to cleanly reuse the enforcement technique, but still requires us to show soundness with respect to our security goals.

### 3.1 Enforcing 1-bit security

We start by showing that with a simple type system, we can make sure that typable programs cannot be used to magnify termination leaks beyond the traditional one-bit limit. The core idea is that the type system prevents information that is a mix of secrets and untrusted inputs from affecting termination behavior, by disallowing it in loop guards.

We equip the set of variables with a function giving the label of each variable,  $label : Vars \rightarrow \mathcal{L}$ . For expressions in general we define the function  $lev$ , assigning each expression with its security level. Function  $lev$  is defined as follows, pattern matching on the form of expression:

$$lev(n) = LH \quad lev(x) = label(x) \quad lev(e_1 \text{ op } e_2) = lev(e_1) \sqcup lev(e_2)$$

While variables have their corresponding label as a level, literals are always low confidentiality and high integrity, as we assume the program source to be public but trusted. Other expressions take the least upper bound of their component levels.

Figure 2 gives the typing relation. The typing context consists only of the level of the program counter,  $pc$ . This level represents expressions on which the control flow context depends, namely `if` and `while` guards. If a command  $c$  is typable under context  $pc$ , written  $pc \vdash c$ , the intention is that  $c$  does not leak when executed, even if the execution itself is conditioned on data of level  $pc$  or higher. Branches of an `if`-command must be typable under the outer  $pc$  level joined with the level of the guard expression (rule IF). The rule ASSIGN uses this to prevent implicit flows: assignments to a variable are only allowed when both the expression and the program counter are below or at the same level as the level of the assigned variable.

The rule WHILE propagates the level of the guard in the same way as IF, but in addition requires that the guard expression joined with the context  $pc$  is strictly below  $HL$ . The intention here is to prevent the attacker from selectively inducing nontermination that depends on the secret.

For example, program 1

```
while  $h$  do skip
```

is typable, because the level of the guard is  $HH$ . As we show below, this implies that it only leaks one bit and the attacker is not able to change termination behavior by varying the public input. Same goes for program

`while  $l$  do skip`

since although the attacker can control termination, it does not reveal anything about the secret. The level of the guard here is  $LL$ . However program 2

`while  $h = l$  do skip`

is not typable, as the level of the guard is  $HL$ . Indeed, recall that the attacker is able to try different inputs until one is found that corresponds to the secret, in which case the whole of  $h$  is revealed.

Our goal is to prove that the type system enforces that programs leak at most one bit (via a termination leak) even in the multi-run setting. To prove that typable programs leak at most one bit, we will start by excluding leaks other than termination leaks. This means that terminating programs satisfy noninterference, i.e., the observable output is independent of the secret input. First, we show that programs typable with a high-confidentiality  $pc$  cannot modify the low output.

**Lemma 1.** *Let  $c$  be a program. If  $HL \vdash c$  or  $HH \vdash c$ , then for any choice of  $v_h, v_l \in D$ , if  $c(v_h, v_l) \neq \perp$  then  $c(v_h, v_l) = v_l$ .*

The proofs of this lemma and other statements can be found in Appendix A.

We now establish noninterference for terminating runs.

**Lemma 2.** *Assuming a typable program  $c$  and ignoring diverging runs,  $c$  satisfies non-interference:*

$$\forall v_h, v_h', v_l \in D : \text{if } c(v_h, v_l) \neq \perp \neq c(v_h', v_l) \quad \text{then} \quad c(v_h, v_l) = c(v_h', v_l).$$

In particular, the above lemma tells us that (ignoring nonterminating runs), the single-run knowledge is unaffected by variation in the secret input. Thus, considering nontermination, the attacker can only observe one of two results, meaning the program only leaks one bit. The following lemma shows that this extends to the multi-run case, by showing that either termination depends only on the secret, or only on the public input. This means that the attacker can not improve their knowledge of  $v_h$  beyond the one bit already leaked, by varying the public input.

**Lemma 3.** *Assume  $c$  is a typable program. Then for arbitrary  $v_h, v_h', v_l, v_l' \in D$  either one of the following condition holds.*

1. *Fixing the secret input, varying the public input reveals nothing:*

$$k_{v_h}(c, v_l) = k_{v_h}(c, v_l')$$

2. *Fixing the public input, varying the secret input reveals nothing:*

$$k_{v_h}(c, v_l) = k_{v_h'}(c, v_l)$$

The basic idea of the proof for this lemma, is that using Lemma 2 and assuming that neither condition holds, we can find a pair of high and low inputs that cause the program to diverge, while either of them can be combined with other inputs to cause the program to terminate successfully. By looking at the guard for the loop that causes divergence, its value must be governed by both high and low data, and so it cannot possibly have been allowed by the type system. Thus the assumption that neither condition holds must be false. As before, the full proof is presented in Appendix A.

We can now use the above results to prove that typable programs leak at most one bit.

**Theorem 1.** *Assume  $c$  is a typable program. Then  $c$  leaks at most one bit, i.e., for all  $v_h \in D$  there are at most two distinct values for  $K_{v_h}$ .*

### 3.2 Enforcing general knowledge policies

We now draw on ideas of delimited release [32] to change our type system so that it enforces a general knowledge policy. Delimited release specifies a declassification policy as a set of expressions called *escape hatches*. Such expressions can refer to secret variables, but their computed values may be assigned to public variables. Thus, an escape hatch defines *what* secret information may be declassified as public. Note that the value of an escape hatch is not released automatically, but the program can use it to compute low confidentiality information that is then released explicitly as the public output.

The knowledge policy, a partition of  $D$ , is specified with an expression  $e_P$ . In terms of delimited release, this expression is an escape hatch, and to focus on the interesting ideas for this paper we assume it is the only one. Since the point of a policy expression is to partition the input space of  $h$ , any useful policy expression will only depend on  $h$ . Thus, we consider escape hatches that only involve high variables and generate policies from escape hatches as follows:

**Definition 7.** *An expression  $e$ , involving no other variables than  $h$ , generates a knowledge policy  $P$  as follows:*

$$P = \{P_1, \dots, P_n\}$$

where for all  $v$  and  $v'$  we have  $e(v) = e(v')$  if and only if  $\llbracket v \rrbracket_P = \llbracket v' \rrbracket_P$ .

In order to support knowledge policies, we extend the type system with the possibility of *declassification*. The escape hatch expression is explicitly declassified to have the level  $LH$ , even though it may involve high confidentiality or low integrity variables. We adapt the definition of  $lev$  accordingly:

$$lev(e) = \begin{cases} LH & \text{if } e = e_P \text{ or } e = n \\ label(x) & \text{if } e \neq e_P \text{ and } e = x \\ lev(e_1) \sqcup lev(e_2) & \text{if } e \neq e_P \text{ and } e = e_1 \text{ op } e_2 \end{cases}$$

The only typing rule that needs to be changed from Figure 2 is the one for assignment, which disallows updates to any variable involved in the escape hatch:

$$\text{ASSIGN} \frac{lev(e) \sqcup pc \sqsubseteq lev(x) \quad x \notin vars(e_P)}{pc \vdash x := e}$$

This is done in order to prevent information about the secret input being laundered through the escape hatch and is standard in delimited release [32]. See Program 7 for an example of laundering.

If the high input is a tuple of multiple high inputs, as described earlier, the ASSIGN rule should simply require that  $x$  is not one of them. We have left it as is in the interest of readability.

We return to the examples of Section 1 to illustrate the soundness and precision of the enforcement. Program 1 is still typable independently of escape hatches. Programs 2 and 3 are rightfully rejected in the absence of escape hatches because they might leak the entire secret. Given the escape hatch  $h\%4$ , the secure programs 4 and 6 are accepted by the type system because declassification relabels  $h\%4$  to  $LH$ , which is under  $LL$  in the lattice, the label of  $l$ . Given the same escape hatch, the insecure program 5 is rejected because  $h\%6$  of type  $HH$  is assigned to variable  $l$  of type  $LL$ . Program 7 is also rejected because variable  $h$  (which is involved in an escape hatch) is modified.

The soundness of the type system is guaranteed by the following theorem.

**Theorem 2.** *Assume  $c$  is a typable program and  $e_P$  is an escape hatch that induces a policy  $P$ . Then  $c$  is 1-bit secure with respect to the policy  $P$ .*

## 4 Related work

Language-based information-flow security is a large and continuously-evolving field [31]. We focus on discussing most related work on knowledge-based security, interactive security, and declassification policies.

*Knowledge-based security* Dima et al. [17] consider sets as representation of attacker’s knowledge in nondeterministic systems. Askarov and Sabelfeld [4] present a knowledge-based condition of *gradual release* for declassification, as well as enforcement for a language with communication primitives. Gradual release allows the knowledge of the attacker to increase only when the program passes a declassification point.

Van der Meyden [38] expresses intransitive noninterference policies using a classical model of knowledge in terms of different agents’ views of the world [18].

Banerjee et al. [7] enhance the knowledge-based representation of attackers with powerful program specification policies. As a result, they are able to express declassification policies of both *what* can be released and *where* in the code.

Askarov and Sabelfeld [5] use knowledge to describe both termination-insensitive and -sensitive security definitions with possibilities of expressing of *what* can be released and *where*, as well as dynamic enforcement for a language with dynamic code evaluation and communication primitives.

Broberg and Sands [11] describe *paralocks*, a knowledge-based framework for expressing versatile declassification policies, including role-based policies.

Demange and Sands [15] allow tuning sensitivity to (non)termination depending on the size of the secret that is involved in loop guards: looping is disallowed when loop guards depend on secrets of small size.

None of the above approaches model the attacker’s knowledge obtained by running the program multiple times.

*Interactive security* Multi-run security is related to interactive security. In particular, multi-run security of a batch-job program  $c$  that operates on secret variable  $h$  and public variable  $l$  can be related to single-run security of the following interactive program:

$$h' := h; \text{while } 1 \text{ do } (\text{in}(l); c; \text{out}(l); h := h')$$

where  $h'$  is an auxiliary variable. This encoding allows us for direct comparison with security definitions of interactive programs.

Le Guernic et al. [22] as well as Askarov and Sabelfeld [4] ignore diverging runs of interactive programs, which, as pointed out previously [7, 2], always allows program like  $c$  in the encoding above to be arbitrarily insecure.

ONEil et al. [27] investigate termination-sensitive security for programs that interact with input/output strategies, where strategies are represented as functions that compute the next input to the program based on the previous communication history. Being termination-sensitive and declassification-free, their condition rejects all of programs 1–7 from Section 1, if plugged to the encoding above.

Clark and Hunt [13] show that for deterministic programs, it makes no difference whether the user is represented by a strategy or an input/output stream. Askarov et al. [2] and Bohannon et al. [10] consider stream-based termination-insensitive security. However, as shown in [2], brute-force attacks similar to programs 2–3 are allowed.

Köpf and Basin [21] propose an information-theoretic model for multi-run security in the context of side-channel attacks. The timing side channel can be thought of as a generalization of the termination channel as nontermination manifests itself as long-lasting computation for a real-world attacker. Their model is based on refining the attacker’s knowledge over multiple runs, well in line with our approach. However, as the motivation of Köpf and Basin’s model is quantitative information leaks, they reason about finite numbers of runs and explore the space between our single-run and multi-run security definitions. Further, their enforcement is of rather different nature from ours: it is based on quantitative approximation using greedy heuristic.

Askarov and Sabelfeld [5] explore stream-based definitions for both termination-insensitive and -sensitive security in the presence of declassification policies. However, similar to the approaches above, the termination-sensitive condition rejects programs all of programs 1–7 and the termination-insensitive condition allows attacks 2–3, when plugged to the encoding above.

We have studied extensions of the multi-run secure type system presented here to interactive programs. Maintaining the 1-bit guarantee of termination-insensitive enforcement across all high inputs is non-trivial, as any public side effect (both input and output) will reveal information about the program counter to an attacker. If such an effect appears after a potentially diverging loop on high data, this will already leak one bit before the program has stopped. We envision that full integration of robust declassification and delimited release for interactive programs might be promising in this direction (see the discussion of dimensions of declassification below), but we expect problems with permissiveness of the enforcement. This indicates a fundamental trade-off between interactivity and security. Our paper identifies a niche, where it is possible to gain permissiveness without sacrificing security.

*Declassification* As mentioned earlier, our declassification policy is an adaptation of *delimited release* [32]. Similarly to Askarov and Sabelfeld [5], we derive knowledge sets from escape-hatch expressions. The treatment of integrity by the type system is inspired by *robust declassification* [41, 25, 3]. Robust declassification guarantees that the attacker may not affect what is released by programs by ensuring that only high-integrity data can be declassified, and only in high-integrity context. In a similar spirit, our type system demands that loop context and guards may not mix high confidential data with attacker-controlled data.

In order to prevent unintended laundering of secrets, delimited release ensures that values of escape-hatch expressions do not change within a single run. In general, this guarantee does not extend over multiple runs, which potentially provides a laundering opportunity if the expression depends on data that is provided by an attacker, or is otherwise non-deterministic between runs. We avoid this issue at its root by not allowing non-secrets in escape hatches.

As we have foreshadowed earlier, we are able to cleanly reuse the robust declassification enforcement technique. However, note that we cannot automatically extract soundness guarantees from soundness results for robust declassification (e.g., [25]). The reason is that robust declassification addresses the *where* dimension of declassification: ignoring exactly *what* is leaked, but making sure the active attacker may not affect the declassification mechanism to leak more than the passive attacker. In contrast, our declassification policies are strict about *what* is leaked: the escape hatches describe the upper bound on leaks in programs.

Other, less related, work on declassification is described in a recent overview of the area [34]. The overview is organized by the dimensions of declassification.

## 5 Conclusions

We have showed how that extremes of insecurity (as with termination-insensitive noninterference) and over-restrictiveness of enforcement (as with termination-sensitive noninterference) can be avoided when generalizing batch-job security to multiple runs. Addressing the problem, we have presented a knowledge-based framework for specifying and enforcing multi-run security policies. The policy framework includes possibilities for declassification. The type-based enforcement tracks both confidentiality and integrity labels and guarantees multi-run security.

We expect interesting implications of our result for multi-threaded programs. The termination channel can be magnified in single-run multi-threaded programs in a fashion similar to using multiple runs of sequential programs. Assume we have as many threads as there are bits in secret  $h$ . Then, the multi-threaded program, where individual thread  $i$  is described as follows

$$T_i : (\text{while } h \&\& b_i \text{ do skip}); \text{out}(i)$$

where  $b_i$  contains all zeros in the boolean representation except for bit  $i$ , leaks the entire secret in a single run. Our type-based enforcement can be straightforwardly applied to prevent this kind of leaks by considering the thread-dependent data  $b_i$  as low integrity. We expect that whenever a collection of threads is typable according to our type system,

then the multi-threaded program that consists of the collection of threads is both single-run and multi-run secure (for a notion of *possibilistic* [24, 37, 33] security suitable for reasoning about nondeterministic programs).

As mentioned earlier, the termination channel can be seen as an instance of the timing side channel as nontermination manifests itself as long-lasting computation for a real-world attacker. We can offer protection against timing attacks that is similar to the protection against termination attacks: when the computation does not mix secret and attacker-controlled data in branch guards, then the timing leaks cannot be magnified. Otherwise, we resort to such existing approaches as cross-copying [1] and predictive black-box mitigation [6].

Note that there is nothing fundamental about our enforcement being static. We expect a dynamic mechanism, such as a monitor for delimited-release like policies by Askarov and Sabelfeld [5] to be easily adaptable to dynamically track both confidentiality and integrity in order to enforce our security condition.

Although the paper operates on a simple two-level security lattice, we do not anticipate difficulties with extending our approach to arbitrary lattices. Requiring the confidentiality level of a loop guard to be bounded by its integrity level gives us a way to prevent the dangerous mix of high-confidentiality and low-integrity data to affect the termination behavior. Other future work focuses on expressing multi-run security for richer languages. Further, we plan to extend the framework to take into account modifications of secret data between program runs. We are also exploring decentralized security policies by knowledge-based representations of multiple attackers.

*Acknowledgments* Thanks are due to John Hughes and Dave Sands for helpful feedback. This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR. Arnar Birgisson is a recipient of the Google Europe Fellowship in Computer Security, and this research is supported in part by this Google Fellowship.

## References

- [1] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.
- [2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, Oct. 2008.
- [3] A. Askarov and A. C. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS, Mar. 2010.
- [4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.
- [5] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
- [6] A. Askarov, D. Zhang, and A. Myers. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security*, pages 297–307, 2010.
- [7] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, May 2008.

- [8] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.
- [9] J. Barnes and J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.
- [10] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, Nov. 2009.
- [11] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2010.
- [12] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.
- [13] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST’08)*, Oct. 2008.
- [14] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
- [15] D. Demange and D. Sands. All secrets great and small. In *Proc. European Symp. on Programming*, volume 5502 of *LNCS*, pages 207–221. Springer-Verlag, 2009.
- [16] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.
- [17] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic noninterference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.
- [18] R. Fagin, J.Y.Halpern, Y.Moses, and M. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
- [19] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.
- [20] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive informationflow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009. Supersedes ISSSE and ISO LA 2006.
- [21] B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. In *ACM Conference on Computer and Communications Security*, pages 286–296, 2007.
- [22] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN’06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.
- [23] G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.
- [24] J. McLean. A general theory of composition for a class of “possibilistic” security properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, Jan. 1996.
- [25] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.
- [26] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at <http://www.cs.cornell.edu/jif>, July 2001.
- [27] K. O’Neill, M. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.
- [28] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.
- [29] J. Rizzo and T. Duong. Padding oracles everywhere. <http://ekoparty.org/juliano-rizzo-2010.php>, 2010.
- [30] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, pages 177–189, July 2006.

- [31] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
- [32] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of LNCS, pages 174–191. Springer-Verlag, Oct. 2004.
- [33] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.
- [34] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, Jan. 2009.
- [35] V. Simonet. The Flow Caml system. Software release. Located at <http://crystal.inria.fr/~simonet/soft/flowcaml>, July 2003.
- [36] G. Smith. On the foundations of quantitative information flow. In *Proc. Foundations of Software Science and Computation Structure*, volume 5504 of LNCS, pages 288–302, Mar. 2009.
- [37] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.
- [38] R. van der Meyden. What, indeed, is intransitive noninterference? In *Proc. European Symp. on Research in Computer Security*, pages 235–250. Springer-Verlag, Sept. 2007.
- [39] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.
- [40] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
- [41] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
- [42] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.
- [43] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.

## Appendix A: Proofs

**Proof of Lemma 1.** By induction on typing derivation,  $c$  contains no assignments to  $l$ , since those are rejected by the constraint on  $pc$  in typing rule ASSIGN. Since nontermination is excluded, the output can only be the initial value of  $l$ , namely  $v_l$ .  $\square$

**Proof of Lemma 2.** We prove the lemma by induction on the structure of the program.

**Case  $c = \text{skip}$**  The statement holds because  $c(v_h, v_l) = v_l$  for any  $v_h, v_l$ .

**Case  $c = x := e$**  The case obviously holds when  $x \neq l$ . Otherwise, since  $c$  is typable under some  $pc$ , we know that  $\text{lev}(e) \sqcup pc \sqsubseteq \text{lev}(l) = LL$ . In particular  $\text{lev}(e)$  is either  $LH$  or  $LL$ . From the definition of  $\text{lev}$  it is easy to show that  $e$  does not contain the high input, so the value of  $e$  must be the same whether the high input is  $v_h$  or  $v_h'$ .

**Case  $c = c_1; c_2$**  Since  $c$  is typable under  $pc$ , then so are  $c_1$  and  $c_2$ . By induction we have that  $c_1(v_h, v_l) = c_1(v_h', v_l)$ , let's call this intermediate low output  $v_l''$ . Then  $c(v_h, v_l) = c_2(v_h, v_l'')$ , which again by induction is equal to  $c_2(v_h', v_l'') = c(v_h', v_l)$ .

**Case  $c = \text{if } e \text{ then } c_1 \text{ else } c_2$**  By induction the theorem holds for both branches  $c_1$  and  $c_2$ . If  $\text{lev}(e)$  is  $LL$  or  $LH$ , the value of  $e$  is unaffected by the secret input being  $v_h$  or  $v_h'$ , so in either case the same branch is selected and the theorem holds.

If  $\text{lev}(e)$  is  $HL$  or  $HH$  then, since both branches are typable under  $pc \sqcup \text{lev}(e)$ , Lemma 1 gives that  $c_i(v_h, v_l) = c_i(v_h', v_l) = v_l$  for both  $i = 1$  and  $i = 2$ .

**Case  $c = \text{while } e \text{ do } c$**  Since we know that  $c$  terminates, we can write any program of this form as a sequence of sufficiently many  $\text{if}$ -statements with the same guard and body. This case can thus be reduced to the cases for conditionals and sequential composition.  $\square$

**Proof of Lemma 3.** Assume that condition 2 does not hold, i.e.,  $k_{v_h}(c, v_l) \neq k_{v_h'}(c, v_l)$  for some choice of  $v_h, v_l$  and  $v_h'$ . Lemma 2 gives that if both terminate, then  $c(v_h, v_l)$  and  $c(v_h', v_l)$  return the same value. Thus there are only two observable outcomes, that value or  $\perp$ . Given our assumption, we thus know that one of the runs diverges. Relabeling if necessary, we can assume that  $c(v_h, v_l) = \perp$ .

Now assume that there is a choice of  $v_l'$  such that condition 1 also does not hold. If we can show this leads to a contradiction, either assumption is false and the theorem holds. Since  $c(v_h, v_l) = \perp$ , this must mean that  $c(v_h, v_l')$  terminates. Based on our assumptions, the facts can be summarized as follows:  $c(v_h, v_l)$  does not terminate;  $c(v_h', v_l)$  terminates; and  $c(v_h, v_l')$  terminates.

Consider the  $\text{while}$  loop that causes  $c(v_h, v_l)$  to diverge, and in particular its guard expression  $e$ . We start by noting that if  $\text{lev}(e) \sqsubseteq HH$  then its valuation can not be affected by the value of  $l$ . This is easy to prove (for expressions in general) from our type system, but laborious so for the sake of our discussion we leave out a precise proof. Symmetrically, if  $\text{lev}(e) \sqsubseteq LL$  then its value can not depend on the value of  $h$ .

The typing rule for loops guarantees that the guard expression is not  $HL$ , so either we have  $e \sqsubseteq HH$  or  $e \sqsubseteq LL$ . In the first case,  $c(v_h, v_l')$  cannot terminate, since  $e$  will always evaluate to the same values (it is evaluated many times) as in  $c(v_h, v_l)$ . In the second case,  $c(v_h', v_l)$  must similarly diverge.

Both cases contradict at least one of our assumptions, so one of them is false.  $\square$

**Proof of Theorem 1.** The proof is by contradiction. Assume that there are three values  $v_{h_1}, v_{h_2}, v_{h_3}$  that would generate three distinct multirun knowledges. If we can show that at least two of  $K_{v_{h_i}}(c)$ , for  $i = 1, 2, 3$ , must coincide then we have the contradiction we need. If  $v_1, \dots, v_n$  is an enumeration of  $D$ , then we can write the three multi-run knowledges as follows.

$$\begin{aligned} K_{v_{h_1}}(c) &= k_{v_{h_1}}(c, v_1) \cap k_{v_{h_1}}(c, v_2) \cap \dots \cap k_{v_{h_1}}(c, v_n) \\ K_{v_{h_2}}(c) &= k_{v_{h_2}}(c, v_1) \cap k_{v_{h_2}}(c, v_2) \cap \dots \cap k_{v_{h_2}}(c, v_n) \\ K_{v_{h_3}}(c) &= k_{v_{h_3}}(c, v_1) \cap k_{v_{h_3}}(c, v_2) \cap \dots \cap k_{v_{h_3}}(c, v_n) \end{aligned}$$

Consider any ‘‘column’’ from this layout, say  $k_{v_{h_i}}(c, v_k)$  for some  $k$  and  $i = 1, 2, 3$ . If the (single-run) knowledges in this column are not all equal, we start by noting that there can be only two possibilities. Lemma 2 gives that if  $c(v_{h_i}, v_k)$  terminates, then the result is unique, so only that value and  $\perp$  can be observed, and thus only two distinct knowledges are possible in the column. Let’s call them  $A$  and  $B$  and say  $k_{v_{h_1}}(c, v_k) =$

$k_{v_{h_2}}(c, v_k) = A$  and  $k_{v_{h_3}}(c, v_k) = B$  (we can rearrange the indexes if necessary). Now Lemma 3 tells us that since the knowledges differ by altering the secret input, they cannot differ by altering the low input. This means that in this case all the single-run knowledges in the same “row” are the same and we obtain that  $K_{v_{h_1}}(c) = K_{v_{h_2}}(c) = A$  and  $K_{v_{h_3}}(c) = B$ . Note that it is enough for only one column to contain different values to force this equality in the rows.

On the other hand, if each column has three equal knowledges, then it is clear that  $K_{v_{h_1}}(c) = K_{v_{h_2}}(c) = K_{v_{h_3}}(c)$ . In either case at least two of them must be equal.  $\square$

**Proof of Theorem 2.** To prove this theorem, we apply the same approach as for Theorem 1. This includes proving a modified version of Lemma 2 where instead of letting high inputs be arbitrary, we restrict them to one indistinguishability class of the policy. We’ll note the use of these lemmas, and how their proofs differ from above, as we use them.

First we observe that during any derivation of the program  $c$ , the variable  $h$  remains constant and equal to  $v_h$ , since the modified type system disallows updates to it. This in turn means, that since  $e_P$  mentions no variables besides  $h$ , that it also remains constant. We can use this to show that ignoring diverging runs,  $c$  is non-interferent when secrets are taken from the same policy class. I.e.

$$\begin{aligned} \forall v_h, v_h', v_l \in D : \\ \text{if } c(v_h, v_l) \neq \perp \neq c(v_h', v_l) \text{ and } \llbracket v_h \rrbracket_{e_P} = \llbracket v_h' \rrbracket_{e_P} \\ \text{then } c(v_h, v_l) = c(v_h', v_l). \end{aligned}$$

where  $\llbracket v \rrbracket_{e_P}$  represents the value of  $e_P$  when the variable  $h$  is assigned the value  $v$ . The notation becomes clear when one considers the fact that this value represents the policy indistinguishability class of  $v$ . The proof of this statement is the same as the proof of Lemma 2, with a minor change: In the case for assignment, one must note that the typing judgement  $LL \vdash x := e_P$  holds, regardless of  $lev(x)$ . This case is harmless, since the extra condition of  $\llbracket v_h \rrbracket_{e_P} = \llbracket v_h' \rrbracket_{e_P}$  provides that the same value will be assigned in both cases. A similar argument must, and can easily be made for Lemma 1.

After establishing this, the rest of the proof follows exactly the same argument as the proof of Theorem 1. We assume towards a contradiction, that there exists three secret inputs  $v_{h_1}, v_{h_2}, v_{h_3}$ , all belonging to the same security class, which give rise to three distinct multi-run knowledges  $K_{v_{h_1}}, K_{v_{h_2}}, K_{v_{h_3}}$ . We arrange their definitions as intersections of single-run knowledges as above, and note that either each single-run knowledge in a particular column is the same (i.e. the observed outcome does not depend on the secret input) or that all single-run knowledges in one line are equal (the observed outcome does not depend on the public input). In the latter case, all multi-run knowledges must be equal, and in the second one the non-interference above for terminating runs thus gives that there can be only two observable outputs.  $\square$

## Appendix B: Multi-run exploit

The following type-safe Jif [26] program contains a termination leak that can be amplified to leak more bits with multiple runs.

```
public class Leaky {
    public Leaky() {}

    public static void main()(principal p, String[]{} args)
        throws (SecurityException)
    {
        int{Alice:} secret = 42;
        int{} input;

        try {
            input = Integer.parseInt(args[0]);
            while (0 != (secret & (1 << input))) { }
        }
        catch (NullPointerException ignored) {}
        catch (ArrayIndexOutOfBoundsException ignored) {}
        catch (NumberFormatException ignored) {}
    }
}
```

A simple Python program can be used to make the 32 invocations of the above program needed to leak the complete secret.

```
import subprocess, time, sys, os

JIF_DIR = "../../jif-3.3.1"

def run_with_timeout(command, timeout):
    proc = subprocess.Popen(command, bufsize=0,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)
    start = time.time()
    while start + timeout > time.time() and proc.poll() is None:
        time.sleep(0.1)

    if proc.poll() is None:
        proc.terminate()
        return None

    out, err = proc.communicate()
    return out, err, proc.returncode

if __name__ == "__main__":
    secret = 0
    for bit in range(32):
        print "Checking bit %d... " % bit,
        r = run_with_timeout(
            ["jif",
             "-classpath", os.path.join(JIF_DIR, "tests"),
             "Leaky", str(bit)],
            1)
        if r is None:
            # This indicates timeout, assume non-termination
            secret = secret | (1 << bit)
            print "1"
        else:
            # This indicates normal termination
            print "0"

    print "Secret is: %d" % secret
```