Asymmetric Secure Multi-Execution with Declassification

Iulia Boloşteanu and Deepak Garg

Max Planck Institute for Software Systems, Germany
{iulia_mb,dg}@mpi-sws.org

Abstract. Secure multi-execution (SME) is a promising black-box technique for enforcing information flow properties. Unlike traditional static or dynamic language-based techniques, SME satisfies noninterference (soundness) by construction and is also precise. SME executes a given program twice. In one execution, called the high run, the program receives all inputs, but the program's public outputs are suppressed. In the other execution, called the low run, the program receives only public inputs and declassified or, in some cases, default inputs as a replacement for the secret inputs, but its private outputs are suppressed. This approach works well in theory, but in practice the program might not be prepared to handle the declassified or default inputs as they may differ a lot from the regular secret inputs. As a consequence, the program may produce incorrect outputs or it may crash. To avoid this problem, existing work makes strong assumptions on the ability of the given program to robustly adapt to the declassified inputs, limiting the class of programs to which SME applies.
To lift this limitation, we present a modification of SME, called asymmetric SME or A-SME. A-SME gives up on the pretense that real programs are inherently robust to modified inputs. Instead, A-SME requires a variant of the original program that has been adapted (by the programmer or automatically) to react properly to declassified or default inputs. This variant, called the low slice, is used in A-SME as a replacement for the original program in the low run. The original program and its low slice must be related by a semantic correctness criteria, but beyond adhering to this criteria, A-SME offers complete flexibility in the construction of the low slice. A-SME is provably sound even when the low slice is incorrect and when the low slice is correct, A-SME is also precise. Finally, we show that if the program is policy compliant, then its low slice always exists, at least in theory. On the side, we also improve the state-of-the-art in declassification policies by supporting policies that offer controlled choices to untrustworthy programs.

1 Introduction

Secure systems often rely on information flow control (IFC) to ensure that an unreliable application cannot leak sensitive data to public outputs. The standard IFC security policy is noninterference, which says that confidential or high inputs must not affect public or low outputs. Traditionally, noninterference and related policies have been enforced using static, dynamic, or hybrid analyses of programs [3], [7], [9], [10], [11], [16], [17], [23], but it is known that such analyses cannot be sound (reject all leaky programs) and precise (accept all non-leaky programs) simultaneously. Secure multi-execution or SME is a promising recent technique that attains both soundness and precision, at the expense of more computational power [13]. Additionally, SME is a *black-box* monitoring technique that does not require access to the program's source code or binary.

Briefly, SME runs two copies of the same program, called high and low, simultaneously. The low run is given only low (public) inputs and its high (secret) outputs are blocked. The high run is given both low and high inputs, but its low outputs are blocked. Neither of the two runs can both see high inputs and produce low outputs, so SME trivially enforces noninterference. Less trivially, it can be shown that if a program is noninterfering semantically, then SME does not change its output behavior, so SME is also precise. SME has been implemented and tested in at least one large application, namely the web browser Firefox [6]. As CPU cores become cheaper, we expect SME to scale better and to be applied to other applications as well.

Whereas SME may sound like the panacea for enforcing noninterference, its deployment in practice faces a fundamental issue: Since the low run cannot be provided high inputs, what must it be provided instead? The original work on SME [13] proposes providing *default values* like 0 or null in place of high inputs. In their seminal work on enforcing declassification policies with SME [26], Vanhoef *et al.* advocate providing *policy-declassified values* in place of high inputs. In either case, the high inputs received by the low run of the program are different from the actual high inputs and may also have different semantics. Consequently, the program must be aware of, and robust to, changes in its high inputs' semantics, otherwise the low run may crash or produce incorrect outputs. This is somewhat contrary to the spirit of SME, which aims to be sound and precise on all (unmodified) programs.

Asymmetric SME (A-SME) The robustness requirement limits the programs to which SME can be applied in practice. To circumvent the limitation, a better solution or method is needed. Such a solution is the primary goal of this paper: We posit a modification of SME, called asymmetric SME or A-SME, that gives up on the SME design of executing the *same* program in the high and low runs. Instead, in A-SME, a second program that has been adapted to use declassified inputs (or default inputs in the degenerate scenario where no declassification is allowed) in place of regular high inputs is used for the low run. This second program, which we call the *low slice*, may be constructed by the programmer or by slicing the original program automatically.

In A-SME, the robustness assumption of SME changes to a semantic correctness criteria on the low slice. This correctness criteria takes the declassification policy into account. We prove three results: (a) Irrespective of the correctness of the low slice, the declassification policy is always enforced by A-SME, (b) If the low slice is correct, then A-SME is precise, and (c) If the original program complies with the declassification policy semantically, then its low slice exists, at least in theory.

Our focus here is on *reactive programs* and declassification policies that are specified separately from the monitored program. The rationale for this focus is straightforward: Both web and mobile applications are inherently reactive and, due to the open nature of the two platforms, applications cannot be trusted to declassify sensitive information correctly in their own code.

Improving expressiveness of policies enforced with SME As a secondary contribution, we improve the expressiveness of declassification policies in existing work on SME with declassification. Specifically, we improve upon the work of Vanhoef *et al.* [26] (VGDPR in the sequel). First, we allow declassification to depend on *feedback* from the program and, second, we allow the sensitivity of an input's presence to depend on policy state. We explain these two points below.

Output feedback. We allow policy state to depend on program outputs. This feedback from the program to the policy permits the policy to offer the program controlled choices in what is declassified, without having to introspect into the state of the program. The following examples illustrate this.

Example 1. Consider a data server, which spawns a separate handler process for every client session. A requirement may be that each handler process declassifies (across the network) the data of at most one client, but the process may choose which client that is. With output feedback, the handler process can produce a special high output, seen only by the SME monitor, to name the client whose data the process wants to access. Subsequently, the policy will deny the low run any data not belonging to that client.

Example 2. Consider an outsourced audit process for income tax returns. A significant concern may be subject privacy. Suppose that the process initially reads non-identifying data about all forms (e.g., only gross incomes and pseudonyms of subjects), and then decides which 1% of the forms it wants to audit in detail. With output feedback, we may enforce a very strong policy without interfering with the audit's functionality: The low run of the audit process can see (and, hence, leak) the detailed data of *only* 1% of all audit forms, but it can choose *which* forms constitute the 1%.

State-dependent input presence. Like some prior work on SME [6], we consider a reactive setting, where the program being monitored reacts to inputs provided externally. In this setting, the mere *presence* of an input (not just its content) may be sensitive. SME typically handles sensitive input presence by not invoking the low run for an input whose presence is high [6], [26]. Generalizing this, our policies allow the decision of whether an input's presence is high to depend on the policy state (i.e., on past inputs and outputs). This is useful in some cases, as the following example demonstrates.

Example 3. Consider a news website whose landing page allows the visitor to choose news feeds from topics like politics, sports, and social, and allows the user to interact with the feed by liking news items. When the user clicks one of these topics, its feed is displayed using AJAX, without navigating the user to another page. On the side, untrusted third-party scripts track mouse clicks for page analytics. A privacy-conscious user may want to hide her interaction with certain feeds from the tracking scripts. For example, the occurrence of a mouse click on the politics feed may be sensitive, but a similar click on the sports feed may not. Thus, the sensitivity of mouse click presence on the page depends on the topic being browsed, making the sensitivity state-dependent.

Contributions To summarize, we make the following contributions:

– We introduce asymmetric SME (A-SME) that uses a program (the low slice) *adapted* to process declassified values in the low run (Section 4). This expands the set of programs on which declassification policies can be enforced precisely using SME.
– We increase the expressiveness of declassification policies in SME, by supporting program feedback and state-dependent input presence (Section 3).
– We prove formally that A-SME enforcement is always secure and, given a correct low slice, also precise (Section 4).
– We show that if the program conforms to the policy then its low slice exists, at least in theory (Section 5).

Proofs and other technical details omitted from this paper are provided in an appendix, available online from the authors' homepages.

Limitations The focus of this paper is on the *foundations* of A-SME; methods for constructing the low slice are left for future work. Also, the *where* dimension of declassification, which allows a program to internally declassify information through special declassify actions, is out of the scope of this work. In the context of SME, the *where* dimension has been studied by VGDPR and independently by Rafnsson and Sabelfeld [21,22] (see Section 6).

2 Programming model

We model *reactive* programs, i.e. programs invoked by the runtime when an *input* is available from the program's environment. In response, the program produces a list of *outputs* and this input-output pattern repeats indefinitely. In processing every input, the program may update its internal *memory* and during the next invocation, the runtime passes the updated memory to the program. This allows past inputs to affect the response to future inputs. Reactive programs are a ubiquitous model of computing and web browsers, servers and OS shells are all examples of reactive programs.

Let Input, Output and Memory denote the domains of inputs, outputs and memories for programs, and let $[\tau]$ denote a finite, possibly empty list of elements of type τ.

Definition 1 (Reactive program). *A reactive program p is a function of type $\mathit{Input} \times \mathit{Memory} \mapsto [\mathit{Output}] \times \mathit{Memory}$.*

$$\frac{}{[], \mu \longrightarrow_p []} R1 \qquad\qquad \frac{p(i,\mu) = (O,\mu') \qquad I, \mu' \longrightarrow_p E}{i :: I, \mu \longrightarrow_p (i,O) :: E} R2$$

Fig. 1: Reactive semantics.

The program p accepts an input and its last memory and produces a list of outputs and an updated memory. We deliberately avoid introducing a syntax for reactive programs to emphasize the fact that A-SME is a black-box enforcement technique that does not care about the syntax of the program it monitors. Concretely, the program p may be written in any programming language with a distinguished syntax for inputs and outputs.

Semantics We use the letters i, I, O and μ to denote elements of Input, [Input], [Output] and Memory. $p(i,\mu) = (O,\mu')$ means that the program p when given input i in memory μ produces the list of outputs O and the new memory μ'. A *run* of the program p, written E, is a finite sequence of the form $(i_1, O_1), \ldots, (i_n, O_n)$. The run means that starting from some initial memory, when the program is invoked sequentially on the inputs $i_1, \ldots, i_n$, it produces the output lists $O_1, \ldots, O_n$, respectively. For $E = (i_1, O_1), \ldots, (i_n, O_n)$, we define its projection to inputs $E|_i = i_1, \ldots, i_n$ and its projection to outputs $E|_o = O_1 +\!\!+ \ldots +\!\!+ O_n$, where $+\!\!+$ denotes list concatenation.

Formally, the semantics of a reactive program p are defined by the judgment $I, \mu \longrightarrow_p E$ (Figure 1), which means that program p, when started in initial memory μ and given the sequence of inputs I, produces the run E. Here, $i :: I$ denotes the list obtained by adding element i to the beginning of the list I. Note that if $I, \mu \longrightarrow_p E$, then $E|_i = I$ and $|E| = |I|$.

3 Declassification policies

Our A-SME monitor enforces an *application-specific* declassification policy. This policy may represent the requirements of the programmer, the site administrator, and the hosting environment, but it must be trusted. We model the policy as an abstract stateful program whose state may be updated on every input and every output. The policy's state is completely disjoint from the monitored program's memory, and is inaccessible to the program directly. In each state the policy optionally produces a *declassified value*, which is made available to the low run of A-SME (the low run does not receive inputs directly). By allowing the policy state (and, hence, the declassified value) to depend on inputs, we allow for policies that, for instance, declassify the aggregate of 10 consecutive inputs, but not the individual inputs, as in the prior work of VGDPR. By additionally allowing the policy state to depend on program outputs, the policy may offer the program choices as explained and illustrated in Section 1, Examples 1 and 2. Finally, as illustrated in Example 3, the policy provides a function to decide whether an input's presence is high or low in a given state.

Definition 2 (Policy $\mathcal{D}$). *A declassification policy $\mathcal{D}$ is a tuple $(S, \mathsf{upd}^i, \mathsf{upd}^o, \sigma, \pi)$, where:*

- *S is a possibly infinite set of states. Our examples and metatheorems often specify the initial state separately.*
- *$\mathsf{upd}^i : S \times Input \to S$ and $\mathsf{upd}^o : S \times [Output] \to S$ are functions used to update the state on program input and output, respectively.*
- *$\sigma : S \to Bool$ specifies whether the* presence *of the last input is low or high. When $\sigma(s) = \mathsf{true}$, the input that caused the state to transition to s has low presence, else it has high presence.*
- *$\pi : S \to Declassified$ is the* projection *or* declassification *function that returns the declassified value for a given state. This value is provided as input to the low run when $\sigma(s) = \mathsf{true}$. Declassified is the domain of declassified values.*

The model of our declassification policies is inspired by the one of VGDPR, but our policies are more general because we allow the policy state to depend on program outputs and to set the input presence sensitivity. While VGDPR consider two declassification functions, one idempotent function for projecting every input to an approximate value, and another one for releasing aggregate information from past inputs, we fold the two into a single function π. See Section 6 for a detailed comparison of our model to VGDPR's model.

Example 4 (Declassification of aggregate inputs). Our first example is taken from VGDPR. A browsing analytics script running on an interactive webpage records user mouse clicks to help the webpage developer optimize content placement in the future. A desired policy might be to prevent the script from recording every individual click and, instead, release the average coordinates of blocks of 10 mouse clicks. Listing 1 shows an encoding of this policy. The policy's internal state records the number of clicks and the sum of click coordinates in the variables cnt and sum, respectively. The policy's input update function upd^i takes the new coordinate x of a mouse click, and updates both cnt and sum, except on every 10th click, when the avg (average) is updated and cnt and sum are reset. The projection function π simply returns the stored avg. Finally, since the last average can always be declassified, the input presence function σ always returns true. The output update function upd^o is irrelevant for this example and is not shown. (As a writing convention, we do not explicitly pass the internal state of the policy to the functions upd^i, upd^o, σ and π, nor return it from upd^i and upd^o. This state is implicitly accessible in the policy's state variables.)

Example 5 (State-dependent input presence). This example illustrates the use of the input presence function σ. The setting is that of Example 3. The policy applies to a news website where the user can choose to browse one of three possible topics: politics, sports, or social. The declassification policy for mouse clicks is the following: On the sports page, mouse clicks are not sensitive; on the social page, the average of 10 mouse click coordinates can be declassified (as in

Listing 1 INPUT AGGREGATION

```
Policy state $s$ (local variables):
  cnt : int
  sum : int
  avg : int
Initialization: $\mathsf{cnt} = 0;\ \mathsf{sum} = 0;\ \mathsf{avg} = 0;$
Update functions:
  $\mathsf{upd}^i(\text{MouseClick } x) =$
    case cnt of
      $\mid 9 \to \{\mathsf{cnt} = 0; \mathsf{avg} = (\mathsf{sum} + x)/10; \mathsf{sum} = 0;\}$
      $\mid \_ \to \{\mathsf{cnt} = \mathsf{cnt} + 1; \mathsf{sum} = \mathsf{sum} + x;\}$
Presence decision function:
  $\sigma() = \mathsf{true}.$
Projection function:
  $\pi() = \mathsf{avg}.$
```

Example 4); on the politics page, not even the existence of a mouse click can be declassified.

Listing 2 shows an encoding of this policy. The policy records the current topic being browsed by the user in the state variable st, which may take one of four values: initial, politics, sports and social. Upon an input (function upd^i), the policy state update depends on st. For st = sports, the click's coordinate x is stored in the variable last_click. For st = social, the policy mimics the behavior of Example 4, updating a click counter cnt, a click coordinate accumulator sum and the average avg once in every 10 clicks. Importantly, when st = politics, the policy state is not updated (the input is ignored). A separate component of upd^i not shown here changes st when the user clicks on topic change buttons.

The input presence function σ says that the input is high when $\mathsf{st} \in \{\mathsf{politics}, \mathsf{initial}\}$ (output is false) and low otherwise. Hence, when the user is browsing politics, not even the presence of inputs is released.

The projection function π declassifies the last click coordinate last_click when the user is browsing sports and the average of the last block of 10 clicks stored in avg when the user is browsing social topics. The value returned by the projection function is irrelevant when the user is browsing politics or has not chosen a topic (because in those states σ returns high), so these cases are not shown.

Example 6 (Output feedback: Data server). This example illustrates policy state dependence on program output, which allows feedback from the program being monitored to the policy. The setting is that of Example 1. A data server handles the data of three clients — Alice, Bob and Charlie. The policy is that the data of at most one of these clients may be declassified by a server process and the process may choose this one client. An encoding of the policy is shown in Listing 3. The policy tracks the process' choice in the variable st, which can take one of the four values: none (choice not yet made), alice, bob or charlie. To make the choice, the process produces an output specifying a user whose data it wants to declassify. The function upd^o records the server's choice in st if the process has

Listing 2 STATE-DEPENDENT INPUT PRESENCE

```
Policy state $s$ (local variables):
  st : {initial, sports, politics, social}
  cnt : int
  sum : int
  last_click : int
Initialization: st = initial;  cnt = 0;  sum = 0;  last_click = 0;
Update functions:
  $\mathsf{upd}^i$(MouseClick $x$) =
    case st of
      | sports → {last_click = $x$; }
      | social →
          case cnt of
            | 10 → {cnt = 1;  sum = $x$; }
            | _ → {cnt = cnt + 1; sum = sum + $x$; }
Presence decision function:
  $\sigma()$ =
    case st of
      | initial → false
      | sports → true
      | politics → false
      | social → case cnt of  | 10 → true  | _ → false.
Projection function:
  $\pi()$ =
    case st of
      | sports → last_click
      | social → sum/10.
```

not already made the choice (upd^o checks that st = none). When user data is read (i.e., a new input from the file system appears), the input update function upd^i compares st to the user whose data is read. If the two match, the read data d is stored in the policy state variable data, else *null* is stored in data. The projection function π simply declassifies the value stored in data.

Example 7 (Output feedback: Audit). This example also illustrates feedback from the program to the policy. The setting is that of Example 2, where an untrusted audit process is initially provided with pseudonyms and non-sensitive information of several client records, and later it identifies a certain fraction of these records, which must be declassified in full for further examination. We have simplified the example for exposition: The audit process reads exactly 100 records and then selects 1 record to be declassified for further examination. Pseudonyms are simply indices into an array maintained by the policy. An encoding of the corresponding policy is shown in Listing 4. The policy variable count counts the number of records fed to the program so far. While count is less than 100, the input update function upd^i simply stores each input record i of five fields in the array records. When count reaches 100, the output update function upd^o allows

Listing 3 Output feedback: Data server

```
Policy state $s$ (local variables):
  $\mathsf{st} : \{\mathsf{none}, \mathsf{alice}, \mathsf{bob}, \mathsf{charlie}\}$
  $\mathsf{data} : \mathsf{file}$
Initialization: $\mathsf{st} = \mathsf{none};\ \mathsf{data} = \mathit{null};$
Update functions:
  $\mathsf{upd}^o(\text{RestrictAccessTo } \mathsf{user}) =$
    if $(\mathsf{st} = \mathsf{none})$ then
      case $\mathsf{user}$ of
        $|$ Alice $\to \{\mathsf{st} = \mathsf{alice};\}$
        $|$ Bob $\to \{\mathsf{st} = \mathsf{bob};\}$
        $|$ Charlie $\to \{\mathsf{st} = \mathsf{charlie};\}$
  $\mathsf{upd}^i(\text{PrivateData } (\mathsf{user}, \mathsf{d})) =$
    if $(\mathsf{st} = \mathsf{user})$ then $\{\mathsf{data} = \mathsf{d};\}$ else $\{\mathsf{data} = \mathit{null};\}$
Presence decision function:
  $\sigma() = \mathsf{true}.$
Projection function:
  $\pi() = \mathsf{data}.$
```

the program to provide a single index idx, which identifies the record that must be declassified in full. The full record stored at this index is transferred to the variable declassified, the array records is erased and count is set to ∞ to encode that the process has made its choice.

The projection function π reveals only the index and the gross income of the last input (at index $(\mathsf{count}-1)$ in records) while count is not ∞. When count has been set to ∞, the single record chosen by the process is revealed in full through the variable declassified.

4 Asymmetric SME

We enforce the declassification policies of Section 3 using a new paradigm that we call asymmetric SME (A-SME). A-SME builds on classic SME, but uses different programs in the high and low runs (hence the adjective asymmetric). Classic SME – as described, for example, by VGDPR – enforces a declassification policy on a reactive program by maintaining two independent runs of the given program. The first run, called the high run, is invoked on every new input and is provided the new input as-is. The second run, called the low run, is invoked for an input only when the input's presence (as determined by the policy) is low. Additionally, the low run is not given the original input, but a projected (declassified) value obtained from the policy after the policy's state has been updated with the new input. Only high outputs are retained from the high run (these are not visible to the adversary) and only low outputs are retained from the low run (these are visible to the adversary). Since the low run sees only declassified values and the high run does not produce low outputs, it must be

Listing 4 OUTPUT FEEDBACK: AUDIT

```
Policy state $s$ (local variables):
  records : array[100] $*$ array[5]
  count : int
  declassified : array[5]
Initialization: records $= null$;  count $= 0$;  declassified $= null$;
Update functions:
  $\mathsf{upd}^i(i) =$
    case count of
      | 100 = return;
      | $x$ = {records[$x$] = $i$; count = $x + 1$; }
  $\mathsf{upd}^o(\mathsf{idx}) =$
    case count of
      | 100 = {declassified = records[idx]; records = $null$; count = $\infty$; }
      | _ = return;
Presence decision function:
  $\sigma() =$ true
Projection function:
  $\pi() =$
    case count of
      | $\infty$ = declassified
      | _ = let (idx, name, address, phone, income) = records[count $- 1$] in (idx, income)
```

the case that the low outputs depend only on declassified values. This enforces a form of noninterference.

The problem with classic SME, which we seek to address by moving to A-SME, is that even though the low and the high runs execute the *same* program, they receive completely different inputs — the high run receives raw inputs, whereas the low runs receives inputs created by the declassification policy. This leads to two problems. First, if the programmer is not aware that her program will run with SME, the low run may crash because it may not be prepared to handle the completely different types of the declassified inputs. Fundamentally, it seems impossible for the program to automatically adapt to the different inputs of the high and the low runs, because it gets no indication of which run it is executing in! Second, if the program tries to enforce the declassification policy internally (which a non-malicious program will likely do), then in the low run, the declassification is applied twice — once by the SME monitor and then internally by the program. In contrast, in a run without SME, the function is applied only once. As a consequence, one must assume that the function that implements declassification is idempotent (e.g., in VGDPR, this declassification function is called "project" and it must be idempotent). These two limitations restrict the scenarios in which SME can be used to enforce declassification policies.

To broaden the scope of enforcement of declassification policies with SME, we propose to do away with requirement that the same program be executed in the high and low runs of SME. Instead, we assume that a variant of the program that has been carefully crafted to use declassified inputs (not the raw inputs)

$$\frac{}{[], s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,\, p_L} []}\text{A-SME-1}$$

$$\frac{\begin{array}{c} s'' = \mathsf{upd}^i(s,i) \qquad \sigma(s'') = \mathsf{false} \\ p(i, \mu_H) = (O, \mu'_H) \qquad s' = \mathsf{upd}^o(s'', O) \qquad I, s', \mu'_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,\, p_L} E \end{array}}{i :: I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,\, p_L} (i, O|_H) :: E}\text{A-SME-2}$$

$$\frac{\begin{array}{c} s'' = \mathsf{upd}^i(s,i) \qquad \sigma(s'') = \mathsf{true} \qquad p_L(\pi(s''), \mu_L) = (O', \mu'_L) \\ p(i, \mu_H) = (O, \mu'_H) \qquad s' = \mathsf{upd}^o(s'', O) \qquad I, s', \mu'_H, \mu'_L \Longmapsto^{\mathcal{D}}_{p,\, p_L} E \end{array}}{i :: I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,\, p_L} (i, O'|_L \mathbin{+\!\!+} O|_H) :: E}\text{A-SME-3}$$

Fig. 3: Semantics of A-SME.

exists. This variant, called the low slice, is used in the low run instead of the original program. The resulting paradigm is what we call asymmetric SME or A-SME. Before delving into the details of A-SME and its semantics, we give an intuition for the low slice.

Low slice. For a program p : Input $\times$ Memory $\mapsto$ [Output] $\times$ Memory, the low slice with respect to policy $\mathcal{D}$ is a program p_L : Declassified $\times$ Memory $\mapsto$ [Output] $\times$ Memory that produces the program's low outputs given as inputs values that have been declassified in accordance with policy $\mathcal{D}$. In other words, the low slice is the part of the program that handles only declassified data.

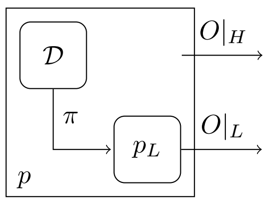


Fig. 2: Factorization of a program p into a declassification policy $\mathcal{D}$ and a low slice p_L.

A question that arises is why this low slice should even exist? Intuitively, if the program p is compliant with policy $\mathcal{D}$, then its low outputs depend only on the output of the policy $\mathcal{D}$. Hence, *semantically*, p must be equivalent to a program that composes $\mathcal{D}$ with some other function p_L to produce low outputs (see Figure 2). It is this p_L that we call p's low slice. We formalize this intuition in Section 5 by proving that if the program p conforms to $\mathcal{D}$ (in a formal sense) then p_L must exist. However, note that the low slice p_L may not be syntactically extractable from the program p by any automatic transformation, in which case the programmer's help may be needed to construct p_L.

4.1 Semantics of A-SME

A-SME enforces a declassification policy $\mathcal{D}$ over a program p and its low slice p_L, together called an *A-SME-aware program*, written (p, p_L). The semantics of A-SME are defined by the judgment $I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,\, p_L} E$ (Figure 3), which should be read: "Starting in policy state s and initial memories μ_H (for the high run) and μ_L (for the low run), the input sequence I produces the run E under A-SME and policy $\mathcal{D}$".

We define the judgment by induction on the input sequence I. Rule A-SME-1 is the base case: When the input sequence I is empty, so is the run E (when there is no input, a reactive program produces no output). Rules A-SME-2 and A-SME-3 handle the case where an input is available. In both rules, the first available input, i, is given to the policy's input update function upd^i to obtain a new policy state s''. Then, $\sigma(s'')$ is evaluated to determine whether the input's presence is high or low (rules A-SME-2 and A-SME-3, respectively).

If the input's presence is high (rule A-SME-2), then only the high run is executed by invoking p with input i. The outputs O of this high run are used to update the policy state to s' (premise $s' = \mathsf{upd}^o(s'', O)$). After this, the rest of the input sequence is processed inductively (last premise). Importantly, any low outputs in O are discarded. The notation $O|_H$ denotes the subsequence of O containing all outputs on high (protected, non-public) channels. We assume that each output carries an internal annotation that specifies whether its channel is high or low, so $O|_H$ is defined.

If the input's presence is low (rule A-SME-3), then in addition to executing the high run and updating the policy state as described above, the low slice p_L is also invoked with the current declassified value $\pi(s'')$ to produce outputs O' and to update the low memory. Only the low outputs in O' ($O'|_L$) are retained. All high outputs in O' are discarded.

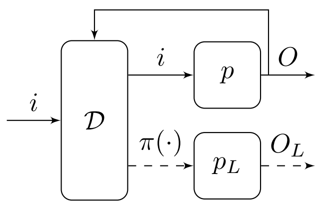


Fig. 4: Pictorial representation of A-SME semantics.

Figure 4 depicts A-SME semantics pictorially. The dashed arrows denote the case where the input's presence is low (A-SME-3). In that case, the low slice executes with the declassified value returned by the policy function π. The arrow from the output O back to the policy $\mathcal{D}$ represents the output feedback.

In the following two subsections we show that A-SME is (1) secure — it enforces policies correctly and has no false negatives, and (2) precise — if p_L is a correct low slice, then its observable behavior does not change under A-SME.

4.2 Security

We prove the security of A-SME by showing that a program running under A-SME satisfies a form of noninterference. Roughly, this noninterference says that if we take two different input sequences that result in the same declassified values, then the low outputs of the two runs of the program under A-SME are the same. In other words, the low outputs under A-SME are a function of the declassified values, so an adversary cannot learn more than the declassified values by observing the low outputs. Importantly, the security theorem makes no assumption about the relationship between p and p_L, so security holds even if a leaky program or a program that does not expect declassified values as inputs is provided as p_L.

$$
\begin{array}{rll}
\mathcal{D}^*(s, []) = [] & & \\
\mathcal{D}^*(s, (i, O) :: E) = \mathcal{D}^*(\mathsf{upd}^o(s'', O), E) & \text{if } s'' = \mathsf{upd}^i(s, i) \text{ and } \sigma(s'') = \mathsf{false} \\
\mathcal{D}^*(s, (i, O) :: E) = \pi(s'') :: \mathcal{D}^*(\mathsf{upd}^o(s'', O), E) & \text{if } s'' = \mathsf{upd}^i(s, i) \text{ and } \sigma(s'') = \mathsf{true}
\end{array}
$$

Fig. 5: Function $\mathcal{D}^*$ returns values declassified by policy $\mathcal{D}$ during a run.

To formally specify our security criteria, we first define a function $\mathcal{D}^*$ (Figure 5) that, given an initial policy state s and a program run E, returns the sequence of values declassified during that run. This function is defined by induction on E and takes into account the update of the policy state due to both inputs and outputs in E. It is similar to a homonym in VGDPR but adds policy state update due to outputs. Note that $\mathcal{D}^*$ adds the declassified value to the result only when the input presence is low (condition $\sigma(s'') = \mathsf{true}$). Equipped with the function $\mathcal{D}^*$, we state our security theorem.

Theorem 1 (Security, noninterference under $\mathcal{D}$) *Suppose $I_1, \mu_1 \longrightarrow_p E_1$ and $I_2, \mu_2 \longrightarrow_p E_2$ and $\mathcal{D}^*(s_1, E_1) = \mathcal{D}^*(s_2, E_2)$. If $I_1, s_1, \mu_1, \mu_L \Longmapsto^{\mathcal{D}}_{p, p_L} E'_1$ and $I_2, s_2, \mu_2, \mu_L \Longmapsto^{\mathcal{D}}_{p, p_L} E'_2$, then $E'_1|_o|_L = E'_2|_o|_L$.*

Proof. By induction on the length of $I_1 +\!\!+ I_2$.

The theorem says that if for two input sequences I_1, I_2, the two runs E_1, E_2 of a program p result in the same declassified values (condition $\mathcal{D}^*(s_1, E_1) = \mathcal{D}^*(s_2, E_2)$), then the A-SME execution of the program on I_1, I_2 will produce the same low outputs ($E'_1|_o|_L = E'_2|_o|_L$) for any low slice p_L. Note that the precondition of the theorem is an equivalence on E_1 and E_2 obtained by execution under standard (non-A-SME) semantics, but its postcondition is an equivalence on E'_1 and E'_2 obtained by execution under A-SME semantics. This may look a bit odd at first glance, but this is the intended and expected formulation of the theorem. The intuition is that the theorem relates values declassified by the standard semantics to the security of the A-SME semantics.

4.3 Precision

In the context of SME, precision means that for a non-leaky program, outputs produced under SME are equal to the outputs produced without SME. In general, SME preserves the order of outputs at a given level, but may reorder outputs across levels. For instance, the rule A-SME-3 in Figure 3 places the low outputs $O'|_L$ before the high outputs $O|_H$. So, following prior work [26], we prove precision with respect to each level: We show that the sequence of outputs produced at any level under A-SME is equal to the sequence of outputs produced at the same level in the standard (non-A-SME) execution. Proving precision for high outputs is straightforward for A-SME.

Theorem 2 (Precision for high outputs) *For any programs p and p_L, declassification policy $\mathcal{D}$ with initial state s, and input list I, if $I, \mu_H \longrightarrow_p E$ and $I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'$, then $E|_o|_H = E'|_o|_H$.*

Proof. From the semantics in Figures 1 and 3 it can be observed that the high run of A-SME mimics (in input, memory and outputs) the execution under $\longrightarrow_p$.

To show precision for low outputs, we must assume that the low slice p_L is *correct* with respect to the original program p and policy $\mathcal{D}$. This assumption is necessary because A-SME uses p_L to produce the low outputs, whereas standard execution uses p to produce them. Recall that the low slice p_L is intended to produce the low outputs of p, given values declassified by policy $\mathcal{D}$. We formalize this intuition in the following correctness criteria for p_L.

Definition 3 (Correct low slice/correct low pair). *A program p_L of type Declassified$\times$Memory $\mapsto$ [Output]$\times$Memory and an initial memory μ_L are called a correct low pair (and p_L is called a correct low slice) with respect to policy $\mathcal{D}$, initial state s, program p and initial memory μ if for all inputs I, if $I, \mu \longrightarrow_p E$ and $\mathcal{D}^*(s, E) = R$ and $R, \mu_L \longrightarrow_{p_L} E'$, then $E|_o|_L = E'|_o|_L$.*

Based on this definition, we can now prove precision for low outputs.

Theorem 3 (Precision for low outputs) *For any programs p and p_L, declassification policy $\mathcal{D}$ with initial state s and input list I, if $I, \mu_H \longrightarrow_p E$ and $I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'$ and (μ_L, p_L) is a correct low pair with respect to $\mathcal{D}$, s, p and μ_H, then $E|_o|_L = E'|_o|_L$.*

The proof of this theorem relies on the following easily established lemma.

Lemma 1 (Low simulation). *Let $I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E$ and $\mathcal{D}^*(s, E) = R$. If $R, \mu_L \longrightarrow_{p_L} E'$, then $E|_o|_L = E'|_o|_L$.*

Proof. By induction on I. Intuitively, the low run in A-SME is identical to the given run under $\longrightarrow_{p_L}$ and the high run of A-SME does not contribute any low outputs.

Proof (of Theorem 3). Let $R = \mathcal{D}^*(s, E')$ and $R, \mu_L \longrightarrow_{p_L} E''$. By Lemma 1, $E'|_o|_L = E''|_o|_L$. From Definition 3, $E|_o|_L = E''|_o|_L$. By transitivity of equality, we get that $E|_o|_L = E'|_o|_L$.

Theorem 4 (Precision) *For any programs p and p_L, declassification policy $\mathcal{D}$ with initial state s and input list I, if $I, \mu_H \longrightarrow_p E$ and $I, s, \mu_H, \mu_L \Longmapsto^{\mathcal{D}}_{p,p_L} E'$, and (μ_L, p_L) is a correct low pair with respect to $\mathcal{D}$, s, p and μ_H, then $E|_o|_L = E'|_o|_L$ and $E|_o|_H = E'|_o|_H$.*

Proof. Immediate from Theorems 2 and 3.

Remark Rafnsson and Sabelfeld [21,22] show that precision across output levels can be obtained for SME using barrier synchronization. We speculate that the method would generalize to A-SME as well.

5 Existence of correct low slices

In this section we show that a correct low slice (more specifically, a correct low pair) of a program exists if the program does not leak information beyond what is allowed by the declassification policy.

Definition 4 (No leaks outside declassification). *A program p starting from initial memory μ does not leak outside declassification in policy $\mathcal{D}$ and initial state s if for any two input lists I_1, I_2: $I_1, \mu \longrightarrow_p E_1$ and $I_2, \mu \longrightarrow_p E_2$ and $\mathcal{D}^*(s, E_1) = \mathcal{D}^*(s, E_2)$ imply $E_1|_o|_L = E_2|_o|_L$.*

Theorem 5 (Existence of correct low slice) *If program p, starting from initial memory μ, does not leak outside declassification in policy $\mathcal{D}$ and initial state s, then there exist p_L and μ_L such that (μ_L, p_L) is a correct low pair with respect to $\mathcal{D}$, s, p and μ.*

We describe a proof of this theorem. Fix an initial memory μ. Define f, g as follows: If $I, \mu \longrightarrow_p E$, then $f(I) = E|_o|_L$ and $g(I) = \mathcal{D}^*(s, E)$. Then, Definition 4 says that $f(I)$ is a function of $g(I)$, meaning that there exists another function h such that $f(I) = h(g(I))$. Intuitively, for a given sequence of declassification values $R = \mathcal{D}^*(s, E)$, $h(R)$ is the set of low outputs of p.

For lists L_1, L_2, let $L_1 \leq L_2$ denote that L_1 is a prefix of L_2.

Lemma 2 (Monotonicity of h). *If $I_1 \leq I_2$, then $h(g(I_1)) \leq h(g(I_2))$.*

Proof. By definition, $h(g(I_1)) = f(I_1)$ and $h(g(I_2)) = f(I_2)$. So, we need to show that $f(I_1) \leq f(I_2)$. Let $\mu, I_1 \longrightarrow_p E_1$ and $\mu, I_2 \longrightarrow_p E_2$. Since $I_1 \leq I_2$, $E_1|_o|_L \leq E_2|_o|_L$, i.e., $f(I_1) \leq f(I_2)$.

We now construct the low slice p_L using h. In the execution of p_L, the low memory μ'_L at any point is the list of declassified values R that have been seen so far. We define:

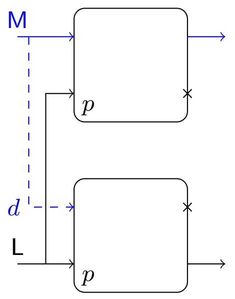
$$\mu_L = []$$
$$p_L(r, R) = (h(R :: r) \setminus h(R), R :: r)$$

If R is the set of declassified values seen in the past, to produce the low output for a new declassified value r, we simply compute $h(R :: r) \setminus h(R)$. By Lemma 2, $h(R) \leq h(R :: r)$ when R and $R :: r$ are declassified value lists from the same run of p, so $h(R :: r) \setminus h(R)$ is well-defined. We then prove the following lemma, which completes the proof.

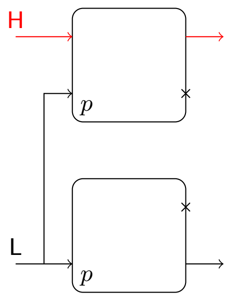
Lemma 3 (Correctness of construction). (μ_L, p_L) *defined above is a correct low pair for $\mathcal{D}$, s, p and μ if p, starting from initial memory μ, does not leak outside declassification in $\mathcal{D}$ and initial state s.*

6 Discussion

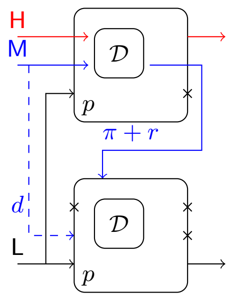
In this section, we compare some of the fine points of A-SME and prior work on SME. We often refer to the schemas of Figure 6, which summarizes several flavors of SME described in the literature.



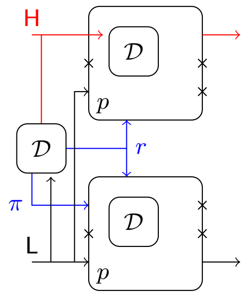
(a) Plain SME [13], 2010.



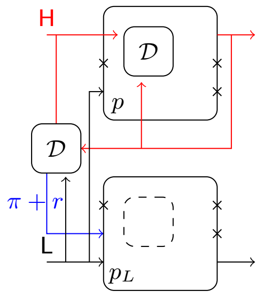
(b) Reactive SME [6], 2011.



(c) Fine-grained SME [21], 2013 (RS).



(d) SME with stateful declassification policies [26], 2014 (VGDPR).



(e) Our A-SME.

Fig. 6: Flavors of SME from literature. Red denotes information at level H, blue denotes information at level M, and black denotes information at level L. *d* is a default value provided to the low run when it demands an input of higher classification.

Input presence levels SME was initially designed by Devriese *et al.* [13] to enforce noninterference on sequential programs, not reactive programs (Figure 6a). They implicitly assume that all inputs are low presence. Thus, there are only two kinds of inputs — low content/low presence (denoted L) and high content/low presence. Following [20], we call the latter "medium"-level or M-level inputs, reserving high (H) for inputs with high presence.

Bielova *et al.* [6] adapted SME for enforcing noninterference in a reactive setting. Though not explicitly mentioned in their paper, their approach assumes that an input's presence and content are classified at the same level. Consequently, in their work, inputs only have levels H and L (Figure 6b). Bielova *et al.* also introduce the idea that for an input with high presence (level H), the low run must not be executed at all and we, as well as VGDPR [26] use this idea. In Bielova *et al.*'s work, an input's presence level is fixed by the channel on which it appears; this static assignment of input presence levels carries into all subsequent work, including that of VGDPR and of Rafnsson and Sabelfeld [21,22] (RS in the sequel). Our work relaxes this idea and permits input presence to depend on policy state.

Input totality RS (Figure 6c) consider all three input levels — L, M, and H — for sequential programs with I/O. In their setup, programs demand inputs and can time how long they wait before an input is available. This allows a conceptual distinction between environments that can always provide inputs on demand and environments that cannot. In an asynchronous reactive setting like ours, VGDPR's, or that of Bielova *et al.*, this distinction is not useful.

Declassification and SME Early work on SME, including that of Devriese *et al.* and Bielova *et al.*, did not consider declassification. RS and VGDPR added support for declassification in the non-reactive and reactive setting, respectively. In RS, declassification policies have two components. A coarse-grained policy, ρ, specifies the flows allowed between levels statically and is enforced with SME. A fine-grained mechanism allows the high run of the program to declassify data to the low run dynamically. This mechanism routes data from a special M-level output of the high run to an M-level input of the low run. This routing is called the *release channel* and is denoted by $\pi + r$ in Figure 6c. Data on the release channel is not monitored by SME and the security theorem for such release is the standard gradual release condition [2], which only says that declassification happens at explicit declassification points of the high run, without capturing *what* is released very precisely. For instance, if Example 4 were implemented in the framework of RS, the only formal security guarantee we would get is that *any* function of the mouse clicks might have been declassified (which is not useful in this example).

In contrast, the security theorem of VGDPR, like ours, captures the declassified information at fine granularity. In VGDPR, policies declassify high inputs using two different functions — a stateless projection function *project*, which specifies both the presence level of an input and a declassified value, and a stateful release function *release* that can be used to declassify aggregate information about past inputs. The output of the projection function (denoted π in Figure 6d) is provided as input to the low run in place of the high input. The decision to pass a projected value to the low run where a high input is normally expected results in problems mentioned at the beginning of Section 4, which motivated us to design A-SME. The output of the release function (denoted r in Figure 6d) is passed along a release channel similar to the one in RS. We find the use of two different channels redundant and thus we combine *release* and *project* into a single policy function that we call π. Going beyond VGDPR, in A-SME, the policy state may depend on program output and the input presence may depend on policy state. As illustrated in Section 3, this allows for a richer representation of declassification policies.

Totality of the monitored program Like VGDPR, we assume that the (reactive) program being monitored is total and terminates in a finite amount of time. This rules out leaks due to the adversary having the ability to observe lack of progress, also called progress-sensitivity [1], [18]. In contrast, RS do not make this termination assumption. Instead, they (meaningfully) prove progress-sensitive noninterference. This is nontrivial when the adversary has the ability to observe termination on the low run, as a scheduler must be chosen carefully. We believe that the same idea can be applied to both VGDPR's and our work if divergent behavior is permitted.

7 Related work

(Stateful) Declassification policies Sabelfeld and Sands [24] survey different methods for representing and enforcing declassification policies and provide a set of four *dimensions* for declassification models. These dimensions — *what*, *where*, *when*, and *who* — have been investigated significantly in literature. Policies often encompass a single dimension, such as *what* in delimited release [23], *where* in gradual release [2], or *who* in the context of faceted values [4], but sometimes also encompass more than one dimension such as *what* and *where* in localized delimited release [3], or *what* and *who* in decentralized delimited release [17]. Our security policies encompass the *what* and *when* dimensions of declassification. We do not consider programs with explicit declassify commands (in fact, we do not consider any syntax for programs) and, hence, we do not consider the *where* dimension of declassification [21], [23], [26].

In the context of security policies specified separately from code, Li and Zdancewic [16] propose relaxed non-interference, a security property that applies to declassification policies written in a separate language. The policies are treated as security levels and enforced through a type system. Swamy *et al.* [25] also define policies separate from the program. They express the policies as security automata, using a new language called AIR (automata for information release). The policies maintain their own state and transition states when a release obligation is satisfied. When all obligations are fulfilled, the automaton reaches an accepting state and performs a declassification. These policies are also enforced using a type system. The language Paralocks [7] also supports stateful declassification policies enforced by a type system. There, the policies are represented as sets of Horn clauses, whose antecedents are called locks. Locks are predicates with zero or more parameters and they exhibit two states: opened (true) and closed (false). The type system statically tracks which locks are open and which locks are closed at every program point. Chong and Myers' conditional declassification policies are similar, but more abstract, and also enforced using a type system [9,10,11]. In the context of SME, Kashyap *et al.* [14] suggest, but do not develop, the idea of writing declassification policies as separate sub-programs. Our work ultimately draws some lineage from this idea.

Secure multi-execution We discussed prior work on SME in Section 6. Here, we mention some other work on related techniques. Khatiwala *et al.* [15] propose *data sandboxing*, a technique which partitions the program into two slices, a private slice containing the instructions handling sensitive data, and a public slice that contains the remaining instructions and uses system call interposition to control the outputs. The public slice is very similar to our low slice, but Khatiwala *et al.* trust the low slice's correctness for *security* of enforcement, while we do not. Nonetheless, we expect that the slicing method used by Khatiwala *et al.* to construct the public slice can be adapted to construct low slices for use with A-SME.

Capizzi *et al.* [8] introduce *shadow executions* for controlling information flow in an operating system. They suggest running two copies of an application with different sets of inputs: a *public copy*, with access to the network, that is supplied dummy values in place of the user's confidential data, and a *private copy*, with no access to the network, that receives all confidential data from the user.

Zanarini *et al.* [28] introduce *multi-execution monitors*, a combination of SME and monitoring, aimed at reporting any actions that violate a security policy. The multi-execution monitor runs a program in parallel with its SME-enforced version. If the execution is secure, the two programs will run in sync, otherwise, when one version performs an action different from the other, the monitor reports that the program is insecure. No support for declassification is provided.

Faceted and sensitive values Faceted values [4] are a more recent, dynamic mechanism for controlling information flow. They are inspired by SME but reduce the overhead of SME by simulating the effect of multiple runs in a single run. To do this, they maintain values for different levels (called facets) separately. For a two-level lattice, a faceted value is a pair of values. Declassification corresponds to migrating information from the high facet to the low facet. We expect that in A-SME, the use of the low slice in place of the original program in the low run will result in a reduction of overhead (over SME), comparable to that attained by faceted values.

Jeeves [27] is a new programming model that uses sensitive values for encapsulating a low- and a high-confidentiality view for a given value. Like faceted values, sensitive values are pairs of values. They are parameterized with a level variable which determines the view of the value that should be released to any given sink. Jeeves' policies are represented as declarative rules that describe when a level variable may be set high or low. The policies enforce data confidentiality, but offer no support for declassification. An extension of Jeeves with faceted values [5] supports more expressive declassification policies, but output feedback is still not supported.

Generic black-box enforcement Remarkably, Ngo *et al.* [19] have recently shown that black-box techniques based on multi-execution can be used to enforce not just noninterference and declassification policies, but a large subset of what are called hyperproperties [12]. They present a generic construction for enforcing any property in this subset. Superficially, their generic construction may look similar to A-SME, but it is actually quite different. In particular, their method would enforce noninterference by choosing a second input sequence that results in the same declassified values as the given input sequence to detect if there is any discrepancy in low outputs. A-SME does not use such a construction and is closer in spirit to traditional SME.

8 Conclusion

This paper introduces asymmetric SME (A-SME) that executes a program and its low slice simultaneously to enforce a broad range of declassification policies. We prove that A-SME is secure, independent of the semantic correctness of the low slice, and also precise when the low slice is semantically correct. Moreover we show that A-SME does not result in loss of expressiveness: If the original program conforms to the declassification policy, then a correct low slice exists. Additionally, we improve the expressive power of declassification policies considered in literature by allowing feedback from the program, and by allowing input presence sensitivity to depend on the policy state.

Future work A-SME can be generalized to arbitrary security lattices. For each lattice level ℓ, a separate projection function π_ℓ could determine the values declassified to the ℓ-run in A-SME. For $\ell \sqsubseteq \ell'$, π_ℓ should reveal less information than $\pi_{\ell'}$, i.e., there should be some function f such that $\pi_\ell = f \circ \pi_{\ell'}$. Additionally, A-SME would require a different slice of the program for every level ℓ.

Another interesting direction for future work would be to develop an analysis either to verify the correctness of a low slice, or to automatically construct the low slice from a program and a policy. Verification will involve establishing semantic similarity of the composition of the low slice and the policy with a part of the program, which can be accomplished using static methods for relational verification. Automatic construction of the low slice should be feasible using program slicing techniques, at least in some cases.

Acknowledgments This work was partially supported by the DFG grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems" (RS^3).

References

1. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Proceedings of the European Symposium on Research in Computer Security (ESORICS). pp. 333–348 (2008)
2. Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: 2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA. pp. 207–221 (2007)
3. Askarov, A., Sabelfeld, A.: Localized delimited release: Combining the what and where dimensions of information release. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security. pp. 53–60. PLAS '07 (2007)
4. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 165–178. POPL '12 (2012)
5. Austin, T.H., Yang, J., Flanagan, C., Solar-Lezama, A.: Faceted execution of policy-agnostic programs. In: Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 15–26. PLAS '13 (2013)

6. Bielova, N., Devriese, D., Massacci, F., Piessens, F.: Reactive non-interference for a browser model. In: 5th International Conference on Network and System Security, NSS 2011. pp. 97–104 (2011)
7. Broberg, N., Sands, D.: Paralocks: Role-based information flow control and beyond. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 431–444. POPL '10 (2010)
8. Capizzi, R., Longo, A., Venkatakrishnan, V.N., Sistla, A.P.: Preventing information leaks through shadow executions. In: Proceedings of the 2008 Annual Computer Security Applications Conference. pp. 322–331. ACSAC '08 (2008)
9. Chong, S., Myers, A.C.: Security policies for downgrading. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004. pp. 198–209 (2004)
10. Chong, S., Myers, A.C.: Language-based information erasure. In: 18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005),. pp. 241–254 (2005)
11. Chong, S., Myers, A.C.: End-to-end enforcement of erasure and declassification. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008. pp. 98–111 (2008)
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008. pp. 51–65 (2008)
13. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: 31st IEEE Symposium on Security and Privacy, S&P 2010. pp. 109–124 (2010)
14. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and termination-sensitive secure information flow: Exploring a new approach. In: 32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA. pp. 413–428 (2011)
15. Khatiwala, T., Swaminathan, R., Venkatakrishnan, V.N.: Data sandboxing: A technique for enforcing confidentiality policies. In: 22nd Annual Computer Security Applications Conference (ACSAC 2006),. pp. 223–234 (2006)
16. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005. pp. 158–170 (2005)
17. Magazinius, J., Askarov, A., Sabelfeld, A.: Decentralized delimited release. In: Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, . Proceedings. pp. 220–237 (2011)
18. Moore, S., Askarov, A., Chong, S.: Precise enforcement of progress-sensitive security. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). pp. 881–893 (2012)
19. Ngo, M., Massacci, F., Milushev, D., Piessens, F.: Runtime enforcement of security policies on black box reactive programs. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015. pp. 43–54 (2015)
20. Rafnsson, W., Hedin, D., Sabelfeld, A.: Securing interactive programs. In: 25th IEEE Computer Security Foundations Symposium, CSF 2012,. pp. 293–307 (2012)
21. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: Fine-grained, declassification-aware, and transparent. In: 2013 IEEE 26th Computer Security Foundations Symposium, 2013. pp. 33–48 (2013)
22. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: Fine-grained, declassification-aware, and transparent. Journal of Computer Security (2015), to appear.

23. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003. pp. 174–191 (2003)
24. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: 18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005),. pp. 255–269 (2005)
25. Swamy, N., Hicks, M.: Verified enforcement of stateful information release policies. In: Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security, PLAS 2008. pp. 21–32 (2008)
26. Vanhoef, M., Groef, W.D., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: IEEE 27th Computer Security Foundations Symposium, CSF 2014. pp. 293–307 (2014)
27. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012. pp. 85–96 (2012)
28. Zanarini, D., Jaskelioff, M., Russo, A.: Precise enforcement of confidentiality for reactive systems. In: 2013 IEEE 26th Computer Security Foundations Symposium, 2013. pp. 18–32 (2013)