Specification and Verification of Side Channel Declassification

Josef Svenningsson and David Sands

Department of Computer Science and Engineering,
Chalmers University of Technology
Göteborg, Sweden
{josefs,dave}@chalmers.se

Abstract. Side channel attacks have emerged as a serious threat to the security of both networked and embedded systems – in particular through the implementations of cryptographic operations. Side channels can be difficult to model formally, but with careful coding and program transformation techniques it may be possible to verify security in the presence of specific side-channel attacks. But what if a program intentionally makes a tradeoff between security and efficiency and leaks some information through a side channel? In this paper we study such tradeoffs using ideas from recent research on declassification. We present a semantic model of security for programs which allow for declassification through side channels, and show how side-channel declassification can be verified using off-the-shelf software model checking tools. Finally, to make it simpler for verifiers to check that a program conforms to a particular side-channel declassification policy we introduce a further tradeoff between efficiency and verifiability: by writing programs in a particular "manifest form" security becomes considerably easier to verify.

1 Introduction

One of the pillars of computer security is confidentiality – keeping secrets secret. Much recent research in *language based security* has focused on how to ensure that information flows within programs do not violate the intended confidentiality properties [SM03]. One of the difficulties of tracking information flows is that information may flow in various indirect ways. Over 30 years ago, Lampson [Lam73] coined the phrase *covert channel* to describe channels which were not intended for information transmission at all. At that time the concern was unintended transmission of information between users on timeshared mainframe computers. In much security research that followed, it was not considered worth the effort to consider covert channels. But with the increased exposure of sensitive information to potential attackers, and the ubiquitous use of cryptographic mechanisms, covert channels have emerged as a serious threat to the security of modern systems – both networked and embedded. The following key papers provide a view of the modern side-channel threat landscape:

Pre-proceedings version, plus appendices. Final version to appear in Proceedings of The Sixth International Workshop on Formal Aspects in Security and Trust (FAST2009), November 2009. Springer-Verlag (LNCS)

- Kocher [Koc96] showed that by taking timing measurements of RSA cryptographic operations one could discover secret keys. Later [KJJ99] it was shown that one could do the same by measuring power consumption.
- Based on Kocher's ideas numerous smart card implementations of cryptographic operations have shown to be breakable. See e.g. [MDS99].
- Brumley and Boneh [BB05] showed that timing attacks were not just relevant to smart cards and other physical cryptographic tokens, but could be effective across a network; they developed a remote timing attack on an SSL library commonly used in web servers.

What is striking about these methods is that the attacks are on the *implementations* and not features of the basic intended functionality. Mathematically, cryptographic methods are adequately secure, but useless if the functionally correct implementation has timing or other side channels.

1.1 Simple Timing Channels

Timing leaks often arise from the fact that computation involves branching on the value of a secret. Different instructions are executed in each branch, and these give rise to a timing leak or a power leak (whereby a simple power analysis [MS00] can reveal information about e.g. control flow paths).

One approach is to ensure that both branches take the same time [Aga00], or to eliminate branches altogether [MPSW05] – an approach that is also well known from real-time systems where it is used to make worst case execution time easy to determine [PB02].

```
1  r = 1;
2  i = m - 1;
3  while (i >= 0) {
4    r = r * r;
5    if (d[i] == 1)
       {
6      r = r * x;
7    }
8    i = i - 1;
9  }
10 return r;
```

Fig. 1. Modular exponentiation

Consider the pseudocode in Figure 1 representing a naïve implementation of modular exponentiation, which we will use as our running example throughout the paper.

The data that goes in to this function is typically secret. A common scenario is that the variable `x` is part of a secret which is to be encrypted or decrypted and variable `d` is the key (viewed here as an array of bits). It is important that these remain secret. (On the other hand, `m`, the length of the key, is usually considered public knowledge.)

However, as this function is currently written it is possible to derive some or all of the information about the key using either a timing or power attack. The length of the loop will always reveal the size of the key – and this is accepted. In the body of the loop there is a conditional statement which is executed depending on whether the current bit in the key is set or not. This means that each iteration of the loop will take different amount of time depending on the value of the key. A timing attack measuring the time it takes to compute the whole result can be used to learn the hamming weight of the key, i.e. the number of 1's. With control over

the key and repeated runs this is sufficient to leak the key [Koc96]. A power analysis could in principle even leak the key in a single run.

1.2 Timing and Declassification

Often the run-time cost of securing an algorithm against timing attacks using a general purpose method is higher than what we are prepared to pay.

```
1 z[0] = r*r mod N
2 z[1] = r*r*M mod N
3 r = z[d[i]]
```

For example, using a table-lookup instead of a branch [Cor99] the conditional can be replaced by the code to the left. This fixes the timing leak, but the algorithm becomes considerably slower – even after eliminating the common subexpression. Another even more costly approach is Agat's *cross copying* idea, whereby (roughly speaking) every branch on a secret value **if** `h` **then** `A` **else** `B` is transformed into **if** `h` **then** `A;[B]` **else** `[A];B` where `[A]` is a ghost copy of `A` which takes the same time to compute but otherwise has no effect. There are optimisations of this approach using unification [KM06], or by making the padding probabilistic [DHW08], but efficiency wise the improvements offered by those techniques are probably not sufficient in this context.

```
1 r = 1;
2 i = m - 1;
3 k = 0;
4 while(i >= 0) {
5   r = r * (k ? x : r);
6   k = k ^ d[i];
7   i = i - (k ? 0 : 1);
8 }
```

Fig. 2. Protected exponentiation

A potential solution to this tension between security and efficiency is to make a tradeoff between the two. For this reason it is not uncommon for algorithms to have *some* side channel leakage. An example of this is the following variation on modular exponentiation, adapted from [CMCJ04], which is intended to provide some (unspecified) degree of protection against simple power analysis attacks (but still leaks the hamming weight of the key).

Research Goals and Approach Our research goal is to determine how to express this tradeoff. There are three key issues to explore:

- Security Policies: how should we specify side-channel declassification?
- Security Mechanisms: how to derive programs which achieve the tradeoff?
- Security Assurance: how can we show that programs satisfy a given policy, with a rigorous specification and formal verification?

This paper deals primarily with the first and the third point.

The first step – a prerequisite to a rigorous specification – is to specify our attacker model. A model sets the boundaries of our investigation (and as always with covert channels, there are certainly attacks which fall outside). We choose (as discussed in Section 3) the *program counter security model* [MPSW05]. This model captures attackers performing simple power and timing analysis.

To specify a security policy we turn to work on declassification. The concept of declassification has been developed specifically to allow the programmer to specify what, where, or when a piece of information is allowed to leak (and by whom). A simple example is a program which requires a password based login. For this program to work it must declassify (intentionally leak) the value of the comparison between the actual and the user supplied password strings.

Declassification has been a recent hot topic in information flow security (see [SS05] for an overview). The standard techniques for declassification seem largely applicable to our problem, but there are some differences. The reason being that (in the context of cryptographic algorithms in particular) we may be interested in the distinction between declassifying some data directly (something which has potentially zero cost to the attacker), and declassifying the data but only through a *side channel* – the latter is what we call *side channel declassification*.

We will adapt existing declassification concepts to specify *what* information we are willing to leak through timing channels (Section 4) . More specifically, we use small programs as a specification of *what* information is leaked. This follows the style of *delimited release* [SM04]. As an example, we might want to specify that a program does not leak more that the hamming weight of the key. This can be achieved by using the program fragment in Figure 3 as a specification: it explicitly computes the hamming weight of the key.

```
1 h = 0;
2 i = m - 1;
3 while (i >= 0) {
4   if (d[i] == 1) {
5     h = h + 1;
6   }
7   i = i - 1;
8 }
```

Fig. 3. Hamming weight computation

The formal definition of side-channel declassification (Section 4) is that if the attacker knows the information leaked by the declassifier then nothing more is learned by running the program.

We then turn to the question of verification. We investigate the use of off-shelf automatic program verification tools to verify side-channel declassification policies. The first step is to reify the side channel by transforming the program to represent the side-channel as part of the program state (Section 5). This reduces the specification of side-channel declassification to an extensional program property.

The next step is to observe that in many common cases we can simplify the side-channel instrumentation. This simplification (described in Section 5.1) does not need to be semantics preserving – it simply needs to preserve the side-channel declassification condition.

As we aim to use automatic off-the-shelf model checkers we need one final transformation to make our programs amenable to verification. We use *self composition* to reduce the verification problem to a safety property of a transformed program. Section 6 describes the approach and experiments with software model checkers.

For various reasons the side-channel declassification property of algorithms can still be hard to verify. The last part of this work (Section 7) introduces a

tradeoff which makes verification much simpler. The idea is to write programs in what we call *manifest form*. In manifest form the program is written in two parts: a declassifier first computes *what* is to be released, and then using this information a side-channel secure program computes the rest.

The verification problem amounts to showing that the second part of the program is indeed side-channel secure (this can be rather straightforward due to the strength of the side-channel security condition), and that the declassifier satisfies the property that it does not leak more through its side channel than it leaks directly. We call these *manifest declassifiers*. Since declassifiers are much simpler (and quite likely useful in many different algorithmic contexts) verification of manifest declassifiers is relatively simple. We show how this technique can overcome the verification limitations of certain verification tools.

2 Preliminaries

In this section we present the language we are going to use and set up the basic machinery in order to define our notion of security.

Since we target cryptographic algorithms we will be using a small while language with arrays. It's syntax is defined below:

$$
\begin{array}{l}
C \in \text{Command} ::= x = e \mid x[y] = e \mid C_1; C_2 \mid \text{if } e \text{ then } C_1 \text{ else } C_2 \mid \text{while } e\ C \mid \text{skip} \\
e \in \text{Expression} ::= x \mid x[e] \mid n \mid e_1 \text{ op } e_2 \mid x\ ?\ y\ :\ z \\
op \in \text{Operators} := + \mid * \mid - \mid \hat{} \mid \text{mod} \mid \ldots
\end{array}
$$

The commands of the program should not require much explanation as they are standard for a small while language.

One particular form of expression that we have chosen to include that may not look very standard (for a toy language) is the ternary operator borrowed from the language C. It's a conditional expression that can choose between the value of two different register based on the value of a third register. We have restricted it to only operate on registers since allowing it to choose between evaluating two general expressions may give rise to side channels. This kind of operation can typically be implemented to take a constant amount of time so that it doesn't exhibit a side channel by using conditional assignment that is available in e.g. x86 machine code.

The semantics of programs is completely standard. We defer the definition until the next section where an operational semantics is given together with some additional instrumentation.

3 Baseline Security Model

In this section we present the semantic security model which we use to model the attacker and to define the baseline notion of declassification-free security. For a good balance between simplicity and strength we adopt an existing approach: the *program counter security model* [MPSW05]. This attacker model is strong

enough to analyze *simple power analysis* attacks [KJJ99] – where the attacker is assumed to be able to make detailed correlations between the power profile of a single run with the instructions executed during that run.

The idea of the *program counter security model* is to assume the attacker can observe a *transcript* consisting of the sequence of program counter positions. This is slightly stronger than an attacker who could perfectly deduce the sequence of instructions executed from a (known) program given a power consumption profile of an execution. It does, however, assume that the power consumption of a particular operation does not depend on the data it manipulates. In particular it does not model differential power analysis.

Suppose a program operates on a state which can be partitioned into a *low* (public) part, and a *high* (secret) part. A program is said to be Transcript-secure if given any two states whose low parts are equal, running the program on these respective states yields equal transcripts and final states which also agree on their low parts.[1]

To specialise this definition to our language we note that it is sufficient for the attacker to observe the sequence of branch decisions in a given run in order to be able to deduce the sequence of instructions that were executed. To this end, in Figure 3 we give an instrumented semantics for our language which makes this model of side channels concrete. Apart from the instrumentation (in the form of labels on the transitions) this is a completely standard small-step operational semantics. The transition labels, o, are either a silent step (τ), a 0 or a 1. A zero or one is used to record which branch was taken in an **if** or **while** statement.

$$\langle n, S\rangle \Downarrow n \qquad \langle x, S\rangle \Downarrow S(x) \qquad \frac{\langle e, S\rangle \Downarrow v}{\langle x[e], S\rangle \Downarrow S(x)(v)}$$

$$\frac{\langle e_1, S\rangle \Downarrow v_1 \quad \langle e_2, S\rangle \Downarrow v_2}{\langle e_1 \text{ op } e_2, S\rangle \Downarrow v_1 \ \overline{\text{op}}\ v_2} \qquad \frac{S(x) \neq 0}{\langle x?y : z, S\rangle \Downarrow S(y)} \qquad \frac{S(x) = 0}{\langle x?y : z, S\rangle \Downarrow S(z)}$$

$$\frac{\langle e, S\rangle \Downarrow v}{\langle x = e, S\rangle \xrightarrow{\tau} \langle \text{skip}, S[x \mapsto v]\rangle} \qquad \frac{\langle e, S\rangle \Downarrow v}{\langle x[y] = e, S\rangle \xrightarrow{\tau} \langle \text{skip}, S[x \mapsto x[S(y) \mapsto v]]\rangle}$$

$$\langle \text{skip}; C, S\rangle \xrightarrow{\tau} \langle C, S\rangle \qquad \frac{\langle C_1, S\rangle \xrightarrow{o} \langle C_1', S'\rangle}{\langle C_1; C_2, S\rangle \xrightarrow{o} \langle C_1'; C_2, S'\rangle}$$

$$\frac{\langle e, S\rangle \Downarrow v \quad v \neq 0}{\langle \text{if } e\ C_1\ C_2, S\rangle \xrightarrow{1} \langle C_1, S\rangle} \qquad \frac{\langle e, S\rangle \Downarrow 0}{\langle \text{if } e\ C_1\ C_2, S\rangle \xrightarrow{0} \langle C_2, S\rangle}$$

$$\frac{\langle e, S\rangle \Downarrow v \quad v \neq 0}{\langle \text{while } e\ C, S\rangle \xrightarrow{1} \langle C; \text{while } e\ C, S\rangle} \qquad \frac{\langle e, S\rangle \Downarrow 0}{\langle \text{while } e\ C, S\rangle \xrightarrow{0} \langle \text{skip}, S\rangle}$$

Fig. 4. Instrumented Semantics

[1] It would be natural to assume that attackers have only polynomially bounded computing power in the size of the high part of the state. For the purposes of this paper our stronger definition will suffice.

Definition 1 (Transcript). *Let $d_1, d_2, \ldots$ range over $\{0,1\}$. We say that a configuration $\langle C, S\rangle$ has a transcript $d_1, \ldots, d_n$ if there exist configurations $\langle C_i, S_i\rangle$, $i \in [1,n]$ such that*

$$\langle C, S\rangle \xrightarrow{\tau}{}^{*} \xrightarrow{d_1} \langle C_1, S_1\rangle \xrightarrow{\tau}{}^{*} \xrightarrow{d_2} \cdots \xrightarrow{\tau}{}^{*} \xrightarrow{d_n} \langle C_n, S_n\rangle \xrightarrow{\tau}{}^{*} \langle \texttt{skip}, S'\rangle$$

for some S'.

In the above case we will write $[\![C]\!]S = S'$ (when we only care about the final state) and $[\![C]\!]^T S = (S', t)$ where $t = d_1, \ldots, d_n$ (when we are interested in the state and the transcript).

For the purpose of this paper (and the kinds of algorithms in which we are interested in this context) we will implicitly treat $[\![C]\!]$ and $[\![C]\!]^T$ as functions rather than partial functions, thus ignoring programs which do not always terminate.

Now we can formally define the baseline security definition, which following [MPSW05] we call Transcript-security:

Definition 2 (Transcript-Security). *Assume a partition of program variables into* low *and* high*. We write $R =_L S$ if program states R and S differ on at most their high variables. We extend this to state-transcript pairs by $(R, t_1) =_L (S, t_2) \iff R =_L S \ \&\ t_1 = t_2$ reflecting the fact that a transcript is considered attacker observable (low).*

A program C is Transcript-secure *if for all R, S, if $R =_L S$ then $[\![C]\!]^T R =_L [\![C]\!]^T S$.*

Note that Transcript-security, as we have defined it, is a very strong condition and also very simple to check. A sufficient condition for Transcript-security is that the program in question (i) does not assign values computed using high variables to low variables, and (ii) does not contain any loops or branches on expressions containing high variables. The main contribution of [MPSW05] is a suite of methods for transforming programs into this form. Unfortunately the transformation can be too costly in general, but that method is nicely complemented by use of declassification.

4 Side Channel Declassification

To weaken the baseline definition of security we adopt one of the simplest mechanisms to specify *what* information may be leaked about a secret: *delimited release* [SM04]. The original definition of delimited release specified declassification by placing declassify labels on various expressions occurring in a program. The idea is that the attacker is permitted to learn about (at most) the values of those expressions in the initial state, but nothing more about the high part of the state.

We will reinterpret delimited release using a simple program rather than a set of expressions. The idea will be to specify a (hopefully small and simple) program D which leaks information from high variables to low ones. A program is Transcript-secure modulo declassifier D if it leaks no more than D, and this leak occurs *through the side channel*.

Definition 3 (Side Channel Declassification). *Let D be a program which writes to variables distinct from all variables occurring in C. We define C to be* Transcript-secure modulo D *if for all R and S such that $R =_L S$ we have*

$$[\![C]\!]R =_L [\![C]\!]S \;\;\&\;\; ([\![D]\!]R = [\![D]\!]S \Rightarrow [\![C]\!]^T R =_L [\![C]\!]^T S).$$

The condition on the variables written by D is purely for convenience, but is without loss of generality. The first clause of the definition says that the only information leak can be through the side channel. The second clause says that the leak is no more than what is directly leaked by D. It is perhaps helpful to consider this clause in contrapositive form: $[\![C]\!]^T R \neq_L [\![C]\!]^T S \Rightarrow [\![D]\!]R \neq [\![D]\!]S$. This means that if there is an observable difference in the transcripts of two runs then that difference is manifest in the corresponding runs of the declassifier. Note that if we had omitted the condition $[\![C]\!]R =_L [\![C]\!]S$ then we would have the weaker property that C would be allowed to leak either through the store or through the side channel – but we wouldn't know which. From an attackers point of view it might take quite a bit more effort to attack a program if it only leaks though the side channel so it seems useful to make this distinction. Clearly there are other variations possible involving multiple declassifiers each leaking through a particular subset of observation channels.

5 Reifying the side channel

In the previous sections we have a definition of security that enables us to formally establish the security of programs with respect to side channel declassification. We now turn to the problem of verifying that particular programs fulfil the security condition. In order to avoid having to develop our own verification method we have chosen to use off-the-shelf software verification tools.

Software verification tools work with the standard semantics of programs. But recall that our security condition uses an instrumented semantics which involves a simple abstraction of side channels. In order to make it possible to use off-the-shelf tools for our security condition we must reify the transcript so that it becomes an explicit value in the program which the tools can reason about. It is easy to see how to do this: we add a list-valued variable t to the program, and transform, inductively, each conditional `if e then C else C'` into `if e then t = t++"1"; C else t = t++"0"; C'` and each while loop `while e do C` into

```
(while e do t = t++"1"; C); t= t++"0"
```

and inductively transform the subexpressions `C` and `C'`.

5.1 Simplifying the instrumentation

Reifying the transcript from the instrumented semantics in this way will create a dynamic data structure (a list) which is not bounded in size in general. Such data structures make programs more difficult to reason about, especially if we

want some form of automation in the verification process. Luckily, there are several circumstances which help us side step this problem. Concretely we use two facts to simplify the reification of the side channel.

The first simplification we use depends on the fact that we do not have to preserve the transcript itself – it is sufficient that it yields the same low-equivalence on programs. Suppose that P^T is the reified variant of the program P and that the reification is through the addition of some low variables. In order to use P^T for verification of side-channel security properties it is sufficient for it to satisfy the following property:

$$\forall R, S. [\![P]\!]^T R =_L [\![P]\!]^T S \iff [\![P^T]\!] R =_L [\![P^T]\!] S$$

We call such a P^T an *adequate* reification of P.

The second simplification that we can perform in the construction of a reified program is that we are specifically targeting cryptographic algorithms. A common structure among the ones we have tried to verify is that the while loops contain straight line code (but potentially conditional expressions). If it is the case that **while** loops don't contain any nested branching or looping constructs then we can avoid introducing a dynamic data structure to model the transcript. Let us refer to such programs as *unnested*. For unnested programs it is simply enough to use one fresh low variable for each occurrence of a branch or loop. Thus the reification transformation for unnested programs is defined by applying the two transformation rules below to each of the loops and branches respectively:

```
r = 1;                                  1
i = m - 1;                              2
k = 0; t = 0;                           3
while(i >= 0) {                         4
  t = t + 1;                            5
  r = r * (k ? x : r);                  6
  k = k xor d[i];                       7
  i = i - (k ? 0 : 1);                  8
}                                       9
```

Fig. 5. Instrumented modular exponentiation

$$\textbf{while } \texttt{e C} \rightsquigarrow \texttt{v = 0; } \textbf{while } \texttt{e (v = v + 1; C)} \qquad (v \text{ fresh})$$

$$\textbf{if } \texttt{e } \textbf{then } \texttt{C } \textbf{else } \texttt{C'} \rightsquigarrow \textbf{if } \texttt{e } \textbf{then } \texttt{v = 1; C } \textbf{else } \texttt{v = 0; C'} \qquad (v \text{ fresh})$$

The program in Figure 5 is an instrumented version of the program in Figure 2. The only change is the new (low) variable `t` which keeps track of the number of iterations in the **while** loop.

6 Self Composition

Standard automatic software model checking tools cannot reason about multiple runs of a program. They deal exclusively with safety properties which involves reasoning about a single run. As is well-known, noninterference properties (like side-channel declassification) are not safety properties – they are defined as properties of pairs of computations rather than individual ones. However, a recent technique has emerged to reduce noninterference properties to safety properties for the purpose of verification. The idea appeared in [DHS03], and was explored

extensively in [BDR04] where the idea was dubbed *self composition*. Suppose C is the program for which we want to verify noninterference. Let θ be a bijective renaming function to a disjoint set of variables from those used in C. Let C_θ denote a variable renamed copy of C. Then the standard noninterference property of C can be expressed as a safety property of $C; C_\theta$ viz. the Hoare triple

$$\{\forall v \in Low.v = \theta(v)\}C; C_\theta\{\forall v \in Low.v = \theta v\}$$

To extend this to deal with side-channel declassification, let us suppose that C^T is an adequate reification of C. Then we can verify Transcript-security modulo D by the Hoare triple above (non side-channel security) in conjunction with:

$$\{\forall v \in Low.v = \theta(v)\}D; D_\theta; C^T; C^T_\theta\{(\forall x \in W.x = \theta(x)) \Rightarrow \forall y \in Low.y = \theta(y)\}$$

where W denotes the variables written by D. Here we take advantage of the assumption that the variables written by D are disjoint from those used in C^T. This enables us to get away with a single renaming. Note that since D is a program and not an expression we cannot simply use it in the precondition of the Hoare triple (c.f. [BDR04,TA05]).

6.1 Experiments using Self Composition

As Terauchi and Aiken discovered when they used self composition, it often resulted in verification problems that were too hard for the model checkers to handle [TA05]. As a result of this they developed a series of techniques for making the result of self composition easier to verify. The main technique is the observation that the low part of the two initial states must be equal and hence any computation that depends only on the low part can safely be shared between the two copies of the program. This was reported to help verifying a number of programs. We employ the same technique in our experiments.

We have used the model checkers Blast[HJMS03] and Dagger[GR06] and applied them to self composed version of the cryptographic algorithms. In particular we have tried to verify the instrumented modular exponentiation algorithm in Figure 5 secure modulo the hamming weight of the key (Figure 3). Appendix A presents the code given to the model checkers. We have also tried all the algorithms proposed in [CMCJ04] since they all exhibit some form of side-channel leak and therefore have to be shown to be secure relative that leak. None of the model checkers were powerful enough to automatically verify the programs secure.

The main reason these tools fail seems to be that they do not reason about the contents of arrays. Being able to reason about arrays is crucial for our running example, as it involves computing the hamming weight of an array.

Another problem comes from the fact that the programs we wish to prove secure may be very different from its declassifier. Relating two different programs with each other is a very difficult task and not something that current software model checkers are designed to do.

By helping the model checkers with some manual intervention it is possible to verify the programs secure. Blast has a feature which allows the user to supply their own invariants. Given the correct invariants it will succeed with the verification. However, these predicates are not checked for correctness and coming up with them can be a highly non-trivial task. We have therefore developed another method for verification which we will explore in the next section.

7 Manifest form

In this section we introduce a new way to structure programs to make verification considerably easier: *Manifest Form*. In manifest form the program is written in two parts: a declassifier first computes *what* is to be released, and then using this information a Transcript-secure program computes the rest. Manifest form represents a tradeoff: writing a program in manifest form may make it less efficient. The idea is that the program makes the declassification explicit in its structure (this is similar to the specification of *relaxed noninterference* [LZ05]). But for this to be truly explicit declassification the declassifier itself should not leak through its side channel – or more precisely, the declassifier should not leak more through its side channel than it does directly through the store.

Definition 4 (Manifest Declassifier). *A program D is said to be a* Manifest Declassifier *if for all R and S*

$$[\![D]\!]S =_L [\![D]\!]R \Rightarrow [\![D]\!]^T S =_L [\![D]\!]^T R$$

As an example of a *non* manifest declassifier, consider the program to the left below which declassifies whether an array of length `m` contains all zeros. Here the array length `m`, and `i` and the declassified value `allz`, are low. This is not manifest because the transcript leaks more than the store: it reveals the position of the first nonzero element. A manifest version of this declassifier is shown on the right:

```
1 i = m - 1; allz = 1;
2 while(allz and i >= 0) {
3   allz = (d[i]? 0 : 1);
4   i = i - 1;
5 }
6 i = 0
```

```
i = m - 1; allz = 1;
while(i >= 0) {
 allz *= (d[i]? 0 : 1);
 i = i - 1;
}
```

Definition 5 (Manifest Form). *A program P is in* Manifest Form *if $P = D; Q$ where D is a manifest declassifier and Q is transcript secure.*

The program in Figure 6 is written in manifest form but otherwise it represents the same algorithm as the program in Figure 2. The first part of the program (lines 1–6) computes the hamming weight of the key, `d`, and this (using low variable `hamming`) is then used in the second part of the program to determine the number of loop iterations.

Another example of a program in manifest form can be found in Appendix B.

```
1  hamming = 0;
2  i = m - 1;
3  while(i >= 0) {
4    hamming += (d[i] ? 1 : 0);
5    i = i + 1;
6  }
```

```
7  r = 1; k = 0;
8  i = m - 1;
9  j = m - 1 + hamming;
10 while(j >= 0) {
11   r = r * (k ? x : r);
12   k = k xor d[i];
13   i = i - (k ? 0 : 1);
14   j = j - 1;
15 }
```

Fig. 6. Modular Exponentiation in Manifest Form

7.1 Manifest Security Theorem

Armed with the definitions of sound manifest declassifiers we can now state the theorem which is the key to the way we verify side-channel declassification.

Theorem 1. *Given a program $P = D; Q$ with D being a sound manifest declassifier and Q is transcript secure then P is transcript secure modulo D*

This theorem helps us decompose and simplify the work of verifying that a program in manifest form is secure. First, showing that Q is transcript secure is straightforward as explained in section 3. Verifying that D is a sound manifest declassifier, which might seem like a daunting task given the definition, is actually something that is within the reach of current automatic tools for model checking. We present the code we used to verify the hamming weight computation in Appendix A.

We apply the same techniques of reifying the side channel and self composition to the problem of verifying sound manifest declassifiers. When doing so we have been able to verify that our implementation of the hamming weight computation in Figure 3 is indeed a sound manifest declassifier and thereby establishing the security of the modular exponentiation algorithm in Figure 6. We have had the same success[2] with all the algorithms presented in [CMCJ04].

8 Related Work

The literature on programming language techniques for information flow security is extensive. Sabelfeld and Myers survey [SM03] although some seven years old remains the standard reference in the field. It is notable that almost all of the work in the area has ignored timing channels. However any automated security checking that does not model timing will accept a program which leaks information through timing, no matter how blatant the leak is.

Agat [Aga00] showed how a type system for secure information flow could be extended to also transform out certain timing leaks by padding the branches of appropriate conditionals. Köpf and Mantel give some improvements to Agat's approach based on code unification [KM06]. In a related line, Sabelfeld and Sands

[2] Using Blast version 2.5

considered timing channels arising from concurrency, and made use of Agat's approach [SS00]. Approximate and probabilistic variants of these ideas have also emerged [PHSW07,DHW08]. The problem with padding techniques in general is that they do not change the fundamental structure of a leaky algorithm, but use the "worst-case principle" [AS01] to make all computation paths equally slow. For cryptographic algorithms this approach is probably not acceptable from a performance perspective.

Hedin and Sands [HS05,Hed08] consider applying Agat's approach in the context of Java bytecode. One notable contribution is the use of a family of *time models* which can abstract timing behaviour at various levels of accuracy, for example to model simple cache behaviour or instructions whose time depends on runtime values (e.g. array allocation). The definitions and analysis are parameterised over the time models. The control flow side channel model [MPSW05] can be seen as an instance of this parameterised model.

More specific to the question of declassification and side channels, as we mentioned above, [DHW08] estimates the capacity of a side channel – something which can be used to determine whether the leak is acceptable – and propose an approximate version of Agat's padding technique. Giacobazzi and Mastroeni [GM05] recently extended the abstract noninterference approach to characterising what information is leaked to include simple timing channels. Their theoretical framework could be used to extend the present work. In particular they conclude with a theoretical condition which, in principle, could be used to verify manifest declassifiers. Köpf and Basin's study of timing channels in synchronous systems[KB06] is the most closely related to the current paper. They study a Per model for expressing declassification properties in a timed setting – an abstract counterpart to the more programmer-oriented delimited release approach used here. They also study verification for deterministic systems by the use of reachability in a product automaton – somewhat analogous to our use of self composition. Finally their examples include leaks of hamming weight in a finite-field exponentiation circuit.

9 Conclusions and Further Work

Reusing theoretical concepts and practical verification tools we have introduced a notion of side channel declassification and shown how such properties can be verified by a combination of simple transformations and application of off-the-shelf software model checking tools. We have also introduced a new method to specify side-channel declassification, *manifest form*, a form which makes the security property explicit in the program structure, and makes verification simpler. We have applied these techniques to verify the relative security of a number of cryptographic algorithms. It remains to investigate how to convert a given program into manifest form. Ideas from [MPSW05,LZ05] may be adaptable to obtain the best of both worlds: a program without the overhead of manifest form, but satisfying the same side-channel declassification property.

References

[Aga00] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.

[AS01] J. Agat and D. Sands. On confidentiality and algorithms. In *Proc. IEEE Symp. on Security and Privacy*, pages 64–77, May 2001.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Journal of Computer and Telecommunications Networking*, 48:701–716, 2005.

[BDR04] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of CSFW'04*, LNCS, pages 100–114. IEEE Press, June 2004.

[CMCJ04] Benot Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-cost solutions for preventing simple sidechannel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.

[Cor99] Jean-Sebastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In C .K. Koc and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 292–302, 1999.

[DHS03] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proc. Workshop on Issues in the Theory of Security*, April 2003.

[DHW08] A. Di Pierro, C. Hankin, and H. Wiklicky. Quantifying timing leaks and cost optimisation. In *10th International Conference on Information and Communications Security, ICICS 2008*, volume 5308 of *LNCS*, pages 81–96. Springer-Verlag, 2008.

[GM05] Roberto Giacobazzi and Isabella Mastroeni. Timed abstract non-interference. In *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings*, volume 3829 of *LNCS*, pages 289–303. Springer-Verlag, 2005.

[GR06] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 2006.

[Hed08] Daniel Hedin. *Program analysis issues in language based security.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2008.

[HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *LNCS*, pages 235–239. Springer-Verlag, 2003.

[HS05] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. In *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '05)*, Electronic Notes in Theoretical Computer Science (to appear), 2005.

[KB06] Boris Köpf and David A. Basin. Timing-sensitive information flow analysis for synchronous systems. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*, volume 4189 of *LNCS*, pages 243–262. Springer-Verlag, 2006.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[KM06] Boris Köpf and Heiko Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[Koc96] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[Lam73] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[LZ05] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.

[MDS99] T.S. Messergers, E.A. Dabbish, and R.H. Sloan. Power analysis attacks on modular exponentiation in smartcards, in cryptographic hardware and embedded systems. In *CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 144–157. Springer Verlag, 1999.

[MPSW05] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[MS00] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *CHES*, pages 78–92, 2000.

[PB02] Peter Puschner and Alan Burns. Writing temporally predictable code. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[PHSW07] Alessandra Di Pierro, Chris Hankin, Igor Siveroni, and Herbert Wiklicky. Tempus fugit: How to plug it. *J. Log. Algebr. Program.*, 72(2):173–190, 2007.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, LNCS. Springer-Verlag, 2004.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, Cambridge, England, 2005. IEEE Computer Society Press.

[TA05] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Static Analysis Symposium*, pages 352–367, 2005.

A Example Code Used in Experiments

In this appendix we present examples of the C code we have given to the software model checkers in our verification experiments. The reason we have used inequalities in the assertions is that the version of Dagger we used did not support equalities.

Listing 1 show the code corresponding to the modular exponentiation algorithm shown in Figure 2, where we have verified transcript-security modulo the hamming weight.

Listing 2 presents the code used to show that the hamming weight computation is a sound manifest declassifier.

```
#include <assert.h>

int modularExponentiation() {

  int m; // Length of the key
  int x1; // The secret 1
  int x2; // The secret 2

  int d1[m]; //Key 1
  int d2[m]; //Key 2

  //Hamming computation 1
  int hamming1 = 0;
  int i = m - 1;
  while(i >= 0) {
    hamming1 += (d1[i] ? 1 : 0);
    i--;
  }

  //Hamming computation 2
  int hamming2 = 0;
  int i = m - 1;
  while(i >= 0) {
    hamming2 += (d2[i] ? 1 : 0);
    i--;
  }

  //PROGRAM COPY 1
  int t1 = 0;
  int r1 = 1;
  int i = m - 1;
  int k = 0;
  while(i >= 0) {
    t1++;
    r1 = r1 * (k ? x1 : r1);
    k = k ^ d1[i];
    i = i - (k ? 0 : 1);
  }

```

```
  //PROGRAM COPY 2
  int t2 = 0;
  int r2 = 1;
  int i = m - 1;
  int k = 0;
  while(i >= 0) {
    t2++;
    r2 = r2 * (k ? x2 : r2);
    k = k ^ d2[i];
    i = i - (k ? 0 : 1);
  }

  if (hamming1 <= hamming2 && hamming1 >= hamming2) {
    assert(t1 <= t2);
    assert(t1 >= t2);
  }

  return r1;
}
```

Listing 1. Verification code for modular exponentiation

```
#include <assert.h>

int hammingManifest() {

  int m; //Length of the key

  int d1[m]; //Key 1
  int d2[m]; //Key 2

  //HAMMING COPY 1
  int i = m - 1;
  int hamming1 = 0;
  while(i >= 0) {
    hamming1 += (d1[i] ? 1 : 0);
    i--;
  }

  //HAMMING COPY 2
  int i = m - 1;
  int hamming2 = 0;
  while(i >= 0) {
    hamming2 += (d2[i] ? 1 : 0);
    i--;
  }

  //HAMMING^T COPY 1
  int i = m - 1;
  int hamming3 = 0;
```

```
  int t1 = 0;
  while(i >= 0) {
    hamming3 += (d1[i] ? 1 : 0);
    i--;
    t1++;
  }

  //HAMMING^T COPY 2
  int i = m - 1;
  int hamming4 = 0;
  int t2 = 0;
  while(i >= 0) {
    hamming4 += (d2[i] ? 1 : 0);
    i--;
    t2++;
  }

  if(hamming1 <= hamming2 && hamming1 >= hamming2) {
    assert(t1 <= t2);
    assert(t1 >= t2);
  }

  return 1;

}
```

Listing 2. Verification code for the hamming weight declassifier

B Another Example of Manifest Form

Here we present another example of an algorithm in manifest form written in C. It is taken from [CMCJ04](figure 4a) and is a two bit sliding window algorithm for modular exponentiation.

```
// m : Length of the key
// x : The secret
// d : The key, padded to length m+1

  // DECLASSIFIER
  int k = 1;
  int i = m;
  int s = 1;
  int l = 0;
  while(i >= 0) {
    k = (s ? 0 : 1) * (k+1);
    s = s ^ d[i+1] ^ (d[i] & (k % 2));
    i = i - k*s - (d[i] ? 0 : 1);
    l = l + 1;
  }

```

```
17
18  //MAIN PROGRAM
19  int R[3];
20
21  R[0] = 1; R[1] = x; R[2] = x * x * x;
22  d[0] = 0;
23  int k = 1;
24  int i = m;
25  int s = 1;
26  int j = l;
27
28  while(j >= 0) {
29    k = (!s) * (k+1);
30    s = s ^ d[i+1] ^ (d[i] & (k % 2));
31    R[0] = R[0] * R[k*s];
32
33    i = i - k*s - (!d[i]);
34    j = j - 1;
35  }
36
37  return R[0];
```