A plausible approach to computer-aided cryptographic proofs

Shai Halevi
IBM T.J. Watson Research Center
shaih@alum.mit.edu

June 15, 2005

Abstract

This paper tries to sell a potential approach to making the process of writing and verifying our cryptographic proofs less prone to errors. Specifically, I advocate creating an automated tool to help us with the mundane parts of writing and checking common arguments in our proofs. On a high level, this tool should help us verify that two pieces of code induce the same probability distribution on some of their common variables.

In this paper I explain why I think that such a tool would be useful, by considering two very different proofs of security from the literature and showing the places in those proofs where having this tool would have been useful. I also explain how I believe that this tool can be built. Perhaps surprisingly, it seems to me that the functionality of such tool can be implemented using only "static code analysis" (i.e., things that compilers do).

I plan to keep updated versions of this docuemnt along with other update reports on the web at `http://www.research.ibm.com/people/s/shaih/CAV/`.

1 Introduction

Most papers describe some work that the authors did (or are currently doing). In contrast, this paper essentially describes some work that I hope someone will do in the future. For the most part, this is a "consumer perspective" on automatic generation/verification of cryptographic proofs. Namely, I looked at proofs from papers in the literature and tried to see what tool would have been helpful in writing and verifying these proofs. Hence, the overall attitude in this paper is looking for ways in which an automated tool can help us write/verify proofs the way we currently have them (as opposed to coming up with new proof frameworks).

The approach that I advocate in this paper can be thought of as a "natural next step" along the way of viewing cryptographic proofs as a sequence of probabilistic games, as articulated by Shoup [Sho04] and Bellare and Rogaway [BR04]. Shoup presents this approach mostly as a conceptual tool to help us organize our arguments and "tame the complexity" of our proofs. Bellare and Rogaway takes this approach one step further, viewing the (pseudo-)code of those games as semi-formal objects. (For example they prove that some specific transformations on the pseudo-code are valid under some specific conditions.) In this paper I propose to go even one step further in this direction, writing the code of those games using a fully-specified programming language and having an automatic tool helping us to manipulate the code and/or check that some code transformations are valid. I will try to argue that writing our proofs in this way adds only a moderate effort on the part of the human prover, but can eliminate many of the common causes for errors in such proofs.

Moreover, I believe that the extra effort on the part of the human prover may be offset by the help that an automated tool can offer in generating a proof.

1.1 Do we have a problem with cryptographic proofs?

Yes, we do. The problem is that as a community, we generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect). I became acutely aware of this when I wrote my EME* paper [Hal04]. After spending a considerable effort trying to simplify the proof, I ended up with a 23-page proof of security for a — err — marginally useful mode-of-operation for block ciphers. Needless to say, I do not expect anyone in his right mind to even read the proof, let alone carefully verify it. (On the other hand I was compelled to write the proof, since it was the only way that I could convince myself that the mode was indeed secure.)

Some of the reasons for this problem are social (e.g., we mostly publish in conferences rather than journals), but the true cause of it is that our proofs are truly complex. After all, the objects that we deal with are non-trivial algorithms, and what we try to prove are some properties of an intricate interaction between a few of these algorithms. Most (or all) cryptographic proofs have a creative part (e.g., describing the simulator or the reduction) and a mundane part (e.g., checking that the reduction actually goes through). It often happens that the mundane parts are much harder to write and verify, and it is with these parts that we can hope to have automated help. In this paper I describe some arguments that belong to the "mundane parts" in many common proofs and can be generated or verified with an automated tool. I also present some guesses as to how such a tool can be built.

1.2 Organization

Section 2 describes some of the typical types of games that we have in cryptographic proofs, as well as some common transformations that we apply to these games. Then I look more closely at two different security proofs, trying to find places where it would have been useful to have some help from an automatic tool. The first (and easier) example is the proof for the mode of operation CMC from my paper with Phil Rogaway [HR03], discussed in Section 3. The second example in Section 4 is the proof of security for the Cramer-Shoup CCA-secure public key encryption (specifically, the proof of the basic scheme from the journal version [CS01, Theorem 6.1]). Then Section 5 discusses some issues concerning how to build such an automated tool.

2 Games in cryptographic proofs

Below I briefly review common types of games that we have in cryptographic proofs, as well as some of the common transformation that we apply to these games. Much of the discussion here is adapted from the work of Bellare and Rogaway [BR04].

A typical game involves a stateful adversary that interacts with some interfaces of whatever scheme that we are dealing with. Since I advocate using an automated tool, the games that I consider here are to be specified using some common programming language, implying that the specification is done at a relatively low level, and must consist of all the details of the game.[1] The only aspect of

[1]Throughout the paper I use pseudo-code to specify games, with the understanding that the automated tool will

```
Some-Game (parameters: a bit b and a bound t on the number of adversary queries):
params ← ...                    // Some parameters of this instance of the scheme
key ← Key-Generation(params)

(type[1], query[1]) ← Adversary(params)
for s ← 1 to t do
      if type[s] = 1 then answer[s] ← I1(s; b, params, key, query[s])
      else answer[s] ← I2(s; b, params, key, query[s])
      if s < t then query[s + 1] ← Adversary(answer[s])

guess ← Adversary(answer[t])
if guess = 1 then output 1, else output 0
```

Figure 1: A sample game that depends on a binary parameter b. The adversary routine is unspecified, but the key-generation and two interfaces **I1**, **I2** should be fully specified.

these games that is left unspecified is the adversary, but everything else is fully specified.

A "canonical" game consists of a main loop, where each iteration calls an *adversary routine*, supplying it with the results of the last iteration and getting back the query to be asked in the current iteration. Then the appropriate interface it invoked for the current query, and the result is again fed to the adversary routine in the next iteration. The game is parametrized by some upper bound on the number of iterations (and maybe also upper bounds on other resources of the adversary). It is usually convenient to assume that the adversary fully consumes its resources, and in particular that the number of iterations of the main loop is always exactly equal to the upper bound. Once the main loop is finished, the game may have some output, and this output is typically just the last thing that the adversary has output (more often than not a single bit). Figure 1 describes (in pseudo-code) a sample game, where an adversary interacts with a scheme that has key-generation and two other interfaces, $I1$ and $I2$.

Analyzing games. The properties that "we care about" in these games are almost always either the probability that the game outputs the bit '1' (to model indistinguishability), or the probability that some "bad events" happen during the execution (to model intractability). Sometimes we are interested in both types of properties, usually when analyzing an intermediate step in the proof.

When dealing with indistinguishability, the goal of the analysis is to compare the probability that two different games output the bit '1'. It is often convenient to specify the two games as a single game that depends on a binary parameter. This is particularly convenient if the two games are very similar. (For example, analyzing the security of an encryption scheme consists of comparing the output from two games that are identical except that the target ciphertext in one is an encryption of m_0 while in the other it is an encryption of m_1.) The sample game in Figure 1 is an example of a game that depends on a binary parameter.

When dealing with intractability, the goal of the analysis is to show that the probability of some bad events is very low. These bad events are defined via *bad-event flags*, that are initialized to false and only have assignments to true in the code. (See more discussion of bad-event flags below, as well as in [BR04].)

deal with a fully specified code which is the equivalent of that pseudo-code.

2.1 Common game transformations

A typical cryptographic proof begins by specifying the game (or games) to be analyzed, and then it goes through a sequence of steps, each time changing some aspects of the current game (and claiming that the change does not substantially alters the behavior of the adversary). In this work I identify the games with their code, and games are manipulated by applying simple transformations to their code.

Permissible transformation. I call a transformation *permissible* if the automatic tool can check that this transformation preserves whatever property that we care about in the current game (maybe up to some small factor). Some examples of permissible transformations follow:

- The simplest transformations are code movement, variable substitution, and elimination of "dead code" (i.e., code that has no effect on the "things that we care about" in the game). As an example of the first two transformations, consider replacing the sequence of three statements $x \leftarrow a$, $y \leftarrow b$, $z \leftarrow x \oplus y$ with the equivalent sequence $y \leftarrow b$, $z \leftarrow a \oplus y$, $x \leftarrow a$. Such transformations "do not change anything in the game", and yet they are often useful. Notice that there already exist automated tools that apply such transformations, namely optimizing compilers.

- The next type of transformations are algebraic manipulations. For example, replacing the sequence $h_1 \leftarrow g^{u_1}$, $h_2 \leftarrow h_1^{u_2}$ by the equivalent sequence $h_1 \leftarrow g^{u_1}$, $h_2 \leftarrow g^{u_1 u_2}$. Automatically verifying that this transformation is permissible requires that the tool knows that g, h_1, h_2 are of type "elements of a prime-order group" and u_1, u_2 are integers (or residues modulo the group order). It also requires that the tool knows about the identities that apply to variables of these types.

  Such knowledge can be obtained by having a "library of permissible transformations" that includes all the commonly used algebraic manipulations. For example, the library will contain the identity $(a^b)^c = a^{bc}$ with some constraints that says to what data-types can this identity be applied.

- The "library of permissible transformations" can also include more complex transformations. For example, it is alway permissible to add a bad-event flag to the code. This corresponds to an argument in a proof where some new condition is designated as a bad event.

  The library will include also a transformation corresponding to the rule "after-bad-is-set-nothing-matters" of Bellare and Rogaway [BR04] (which is essentially the same as Shoup's "forgetful gnome" [Sho04]). This rule says that one can freely modify any piece of code that is guaranteed to be executed only when some bad-event flag is set. For example, the statement "**if** bad = true **then** $X \leftarrow 2$" can be replaced by "**if** bad = true **then** $Y \leftarrow 7$." (The justification to this rule is that eventually the proof will show that the bad-event flag is almost never set, so the changes will almost never matter.)

  Yet another permissible transformation (under some conditions) is the coin-fixing technique from [BR04, Section 4.2]. For more details, see discussion of the game "NON1" on Page 12, and also the discussion of coin-fixing in Section 5.

Eliminating bad-event flags. In addition to modifying the games in "ways that do not matter" as above, the tool can be used also to verify arguments that do matter in the analysis. In particular, many of the arguments that are employed to argue that bad events rarely happen can be represented in ways that a simple automated tool can verify. As an easy example, consider the following piece of code (in which the variable flag is designated as a bad-event flag):

00 $\quad \mathsf{flag} \leftarrow \mathsf{false},\ \ X \leftarrow \text{some-}n\text{-bit-string}$
01 $\quad Y \xleftarrow{\$} \{0,1\}^n$
02 $\quad$ **if** $X = Y$ **then** $\mathsf{flag} \leftarrow \mathsf{true},\ \ldots$

After the assignment on line 01, the tool knows that Y is a variable of type "random n-bit string". It can also check that the value of X is independent of Y in the sense of control-flow of the program. Namely, if you draw a graph with all the variables as nodes and an edge $V_1 \longrightarrow V_2$ when the value of V_1 is used in an assignment to V_2, then there is no node V in the graph from which both X and Y are reachable.

Assume that the tool is endowed with a rule that says that when V is a random n-bit string and V' is independent of V in a control-flow sense, then $V = V'$ has probability at most $1/2^n$. In particular the tool can check that the condition $X = Y$ in line 02 has probability at most $1/2^n$. The user can therefore tell the tool to remove the designation of the variable flag as a bad-event flag, and the tool can automatically compute a "penalty" of $1/2^n$ for this modification. Moving to a slightly more involved example, suppose that the code from above was executed inside a loop, and that the loop repeated at most t times (where t is a fixed parameter). In this case, the tool will set the penalty for removing the designation of flag as a bad-event flag to be $t/2^n$.

In general, whenever the user tells the tool to remove the bad-event designation of some flag, the tool can check if it has some rule that bounds the probability of that flag being set. If such a rule is found, the tool can automatically record the penalty that is associated with it. Otherwise, the tool marks this modification as "impermissible" (which means that the user will have to prove on paper that the probability of that flag being set is indeed small, see discussion later in this section).

Notice that removing the bad-event designation of flag does not by itself modify the code in any way. Rather, this is a way for the user to tell the tool that "we no longer care about the variable flag." If flag is never used for other purposes, it then becomes "dead code" and can be removed.

Eliminating the output. Other than bad-event flags, we may also "care about" the output of a game. In particular, proofs of indistinguishably have two different games (or a game that depends on a binary parameter) and the assertion that needs to be proven is that they both output '1' with about the same probability.

As with bad events, here too it is often possible to represent the indistinguishably argument in a way that the automatic tool can verify. In particular, suppose that the analysis starts with a game that depends on a binary parameter, and after a few modifications it arrives at a game in which the output no longer depends on the binary parameter. Then eliminating the output becomes a permissible transformation (without any penalty). Similarly, starting from two games G_0 and G_1, if it is possible to modify both until they "meet" at a common game G', then eliminating the output from G' is permissible.

Using reductions. The transformations that I discussed above are all "unconditionally justified". In most proofs, however, some of the transformations will only be justified by means of a reduction. I believe that it is possible to represent most of our reductions so that the tool can automatically verify them. (In fact, I think that the most important contribution of an automated tool is in its ability to verify reduction steps.) This is done by representing the reduction itself as a game sequence involving permissible transformations. Two examples of such reductions steps can be found in the discussion of the Cramer-Shoup security proof in Section 4.

Impermissible transformations. The user can always choose to perform "impermissible transformations", which are simply transformations for which the analysis is not done via the tool. For example, there are cases where there are two different bad events, but due to symmetry it is enough to analyze only one of them. The tool can be told to drop the bad-event designation of one of the flags, but it seems very hard to automatically verify the effect of this change.

When the user applies a transformation that "the tool does not understand", the tool will simply record the fact that this is an impermissible transformation and will allow the user to specify free-text justification for it. When the proof is verified, the human verifier will be given that free-text explanation (that would presumably refer to some lemma in the proof as it appears on ePrint).

2.2 Structure of typical proofs

A typical cryptographic proof begins with the specification of (a) the scheme to analyze, and (b) the security notion to be proven. (For example, the user may specify some encryption scheme and wish to prove that it is CPA-secure.) These specified, it should be possible to automatically generate the code for the game (or games) that require analysis.

If the security notion is of the intractability variety then there will be a single game with some bad event, and the goal of the analysis is to prove that this bad event rarely happens. From the tool's perspective, the bad event is represented by a bad-event flag, and the goal is to eliminate that flag while incurring only a small penalty. When proving indistinguishability, there are two games that have outputs (or a game that depends on a binary parameter), and the goal is to prove that they output '1' with about the same probability. Again, from the tool's perspective out goal is to eliminate the output while incurring only a small penalty. Either way, from the tool's perspective the goal is to arrive at the empty game (where all the things that "we care about" are already analyzed and there is nothing left in the code to analyze).

The proof will proceed by applying some transformations to the game(s) at hand, until they can be reduced to the empty game. The user will typically add some new bad-event flags to the code (because the analysis identifies some bad events), and then will add or remove pieces of code that are only executed when these bad-event flags are set. The user will then also apply some reduction steps, where the code transformation is justified by a reduction to some computational assumption. Eventually the code is morphed until some of of the built-in rules in the tool can be applied to eliminate the bad events and the output.

If the user can arrive at the empty game while only using permissible transformations, then the proof of security is completely verified by the automated tool. (This means that barring bugs in the tool, the scheme is indeed provably secure under the stated assumptions.) Otherwise, the user may employ some impermissible transformations, in which case some of the analysis would have to be done on paper.

Notice that the way I described things, the tool is always used to do the game transformations, so it has a complete picture of all the steps in the proof, and each step is tagged as either permissible (maybe with some penalty) or impermissible (with a free-text justification). The tool can therefore output some graphical representation of the proof, say in the form of a directed graph that describes what games were used and what type of transformations were applied. Such graphical representation can be helpful to a human verifier who wants to either verify the impermissible transformations, or to "understand the ideas in the proof". For example, Figures 5 and 14, respectively, have graphical representations of the proofs of security for the CMC mode of operation and the Cramer-Shoup cryptosystem.

3 The CMC mode of operation

In this section I discuss the proof of the CMC mode of operation [HR03], as an example of an information-theoretic proof that can benefit from having an automated tool. The CMC mode of operation turns an n-bit block cipher into a "tweakable enciphering scheme" for strings of mn bits, where $m \geq 2$. As in pretty much all modes of operation, proving security is done by modeling the underlying block cipher as a truly random permutation on n-bit strings, and considering an adversary A that can ask to encrypt or decrypt at most q messages, each of m blocks (where q, m are parameters). Then the proof shows (essentially) that the view of A when interacting with the CMC mode differs by at most $O(q^2 m^2 / 2^n)$ from its view when interacting with a process that returns just independent uniformly random bits.

Like many other proofs for modes of operation, the security proof of CMC consists of two parts: First comes a *game-substitution argument*, whose goal is "to simplify the rather complicated setting of A adaptively querying its oracles and to arrive at a simpler setting where there is no adversary and no interaction". Namely, to replace the probabilistic game describing the interaction of the attacker A with the CMC mode, with a different probabilistic game that is non-interactive. That game-substitution argument takes about seven pages, and it is followed by an analysis of the final non-interactive game (essentially a case analysis of possible events) that takes five pages.

The case-analysis at the end embodied "the real reason" for the security of the mode, as it essentially lists all the possible attacks and verifies that they all fail. Although tiresome, it is exactly the part that describe the "innovative" portion of the work (i.e., why is it essential to have all the operations that are specified in the mode), and I do not see how an automated tool can play a significant role in it. On the other hand, the game-substitution argument is quite mechanical, and conveys no intuition about the mode and its security. It is this argument where I think that an automated tool would be most useful. Also, as I describe next, I think that automated help with this argument is doable.

3.1 Using the tool for the CMC proof

Before proving security, the user must describe to the tool what is the construction, and what needs to be proven about it. In the case of CMC, the user will launch the tool and open a template for "modes of operation for a block cipher". This template lets the user specify an enciphering and deciphering routines, with subroutine calls to procedures for block-encryption and block-decryption. (The code for the block cipher will not be specified, since it is viewed as a black box.) The code of CMC is specified in Figure 2.

Algorithm $E_K^T(P_1 \cdots P_m)$

$\mathbf{T}^* \leftarrow E_K(T)$
$PPP_0 \leftarrow \mathbf{T}^*$
for $i \leftarrow 1$ **to** m **do**
    $PP_i \leftarrow P_i \oplus PPP_{i-1}$
    $PPP_i \leftarrow E_K(PP_i)$

$M \leftarrow 2\,(PPP_1 \oplus PPP_m)$
for $i \in [1 \mathinner{..} m]$ **do**
    $CCC_i \leftarrow PPP_{m+1-i} \oplus M$

$CCC_0 \leftarrow 0^n$
for $i \in [1 \mathinner{..} m]$ **do**
    $CC_i \leftarrow E_K(CCC_i)$
    $C_i \leftarrow CC_i \oplus CCC_{i-1}$

$C_1 \leftarrow C_1 \oplus \mathbf{T}^*$
return $C_1 \cdots C_m$

Algorithm $D_K^T(C_1 \cdots C_m)$

$\mathbf{T}^* \leftarrow E_K(T)$
$CCC_0 \leftarrow \mathbf{T}^*$
for $i \leftarrow 1$ **to** m **do**
    $CC_i \leftarrow C_i \oplus CCC_{i-1}$
    $CCC_i \leftarrow E_K^{-1}(CC_i)$

$M \leftarrow 2\,(CCC_1 \oplus CCC_m)$
for $i \in [1 \mathinner{..} m]$ **do**
    $PPP_i \leftarrow CCC_{m+1-i} \oplus M$

$PPP_0 \leftarrow 0^n$
for $i \in [1 \mathinner{..} m]$ **do**
    $PP_i \leftarrow E_K^{-1}(PPP_i)$
    $P_i \leftarrow PP_i \oplus PPP_{i-1}$

$P_1 \leftarrow P_1 \oplus \mathbf{T}^*$
return $P_1 \cdots P_m$

Figure 2: Enciphering (left) and deciphering (right) under $CMC[E]$, where E is a block cipher. The plaintext is $P = P_1 \cdots P_m$ and the ciphertext is $C = C_1 \cdots C_m$.

Once the code for the mode of operation is specified (including all the relevant parameters), the tool can be used to (a) output a "C" program that implements that code (that can be used, say, to generate test vectors); (b) output a LaTeX pseudo-code to embed in the next CRYPTO submission; and (c) move to a "proof of security" mode.

In the "proof of security" mode, the tool can automatically add a main loop that repeatedly calls an unspecified adversary routine A, giving it the output from the last call to the enciphering/deciphering routines and get from it the input to the next call. It also replaces every internal variable in these routines with an array that stores the values that this variable is assigned in the different calls. (In this section, I denote this array by adding a subscript s for every variable. For example PPP_i^s instead of PPP_i.) Next, the tool adds the function π and the sets Domain and Range, initializes them to undefined and empty, respectively, and replaces every occurrence of $Y \leftarrow E(X)$ with the the following piece of code:

1. $Y \xleftarrow{\$} \{0,1\}^n$
2. **if** $Y \in \text{Range}$ **then** $Y \xleftarrow{\$} \overline{\text{Range}}$
3. **if** $X \in \text{Domain}$ **then** $Y \leftarrow \pi(X)$
4. $\pi(X) \leftarrow Y$, $\text{Domain} \leftarrow \text{Domain} \cup \{X\}$, $\text{Range} \leftarrow \text{Range} \cup \{Y\}$

(This code is perhaps awkward-looking, but it turns out to be the most useful way of specifying an on-the-fly choice of permutation for analysis of modes of operation.) A similar piece of code is generated for every occurrence of $X \leftarrow E^{-1}(Y)$.

In addition to the game describing the attack on CMC, the tool will also generate a game where each output block is replaced by a random block, chosen uniformly from $\{0,1\}^n$. In either code, the adversary makes exactly q queries, and then output some bit, and the output of the game is the bit that the adversary outputs. The goal of the human prover is then to prove that the adversary outputs '1' in both games with about the same probability.

Initialization:

$\mathsf{bad} \leftarrow \mathsf{false}$; Domain $\leftarrow$ Range $\leftarrow \emptyset$; **for** all $X \in \{0,1\}^n$ **do** $\pi(X) \leftarrow \mathsf{undef}$

Respond to the s-th adversary query as follows:

An encipher query, $\mathsf{Enc}(P_1^s \cdots P_m^s)$:

Let $u[s]$ be the largest value in $[0\,..\,m]$ s.t. $P_1^s \cdots P_{u[s]}^s = P_1^r \cdots P_{u[s]}^r$ for some $r < s$
$PPP_0^s \leftarrow CCC_0^s \leftarrow 0^n$; **for** $i \leftarrow 1$ **to** $u[s]$ **do** $PP_i^s \leftarrow P_i^s \oplus PPP_{i-1}^s$, $PPP_i^s \leftarrow PPP_i^r$
for $i \leftarrow u[s]+1$ **to** m **do**
    $PP_i^s \leftarrow P_i^s \oplus PPP_{i-1}^s$
    $PPP_i^s \xleftarrow{\$} \{0,1\}^n$; **if** $PPP_i^s \in$ Range **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $PPP_i^s \xleftarrow{\$} \overline{\text{Range}}$
    if $PP_i^s \in$ Domain **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $PPP_i^s \leftarrow \pi(PP_i^s)$
    $\pi(PP_i^s) \leftarrow PPP_i^s$, Domain $\leftarrow$ Domain $\cup \{PP_i^s\}$, Range $\leftarrow$ Range $\cup \{PPP_i^s\}$
$M^s \leftarrow 2\,(PPP_1^s \oplus PPP_m^s)$; **for** $i \in [1\,..\,m]$ **do** $CCC_i^s \leftarrow PPP_{m+1-i}^s \oplus M^s$
for $i \leftarrow 1$ **to** m **do**
    $CC_i^s \xleftarrow{\$} \{0,1\}^n$; **if** $CC_i^s \in$ Range **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $CC_i^s \xleftarrow{\$} \overline{\text{Range}}$
    if $CCC_i^s \in$ Domain **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $CC_i^s \leftarrow \pi(CCC_i^s)$
    $C_i^s \leftarrow CC_i^s \oplus CCC_{i-1}^s$
    $\pi(CCC_i^s) \leftarrow CC_i^s$, Domain $\leftarrow$ Domain $\cup \{CCC_i^s\}$, Range $\leftarrow$ Range $\cup \{CC_i^s\}$
return $C_1 \cdots C_m$

A decipher query, $\mathsf{Dec}(C_1^s \cdots C_m^s)$:

Let $u[s]$ be the largest value in $[0\,..\,m]$ s.t. $C_1^s \cdots C_{u[s]}^s = C_1^r \cdots C_{u[s]}^r$ for some $r < s$
$CCC_0^s \leftarrow PPP_0^s \leftarrow 0^n$; **for** $i \leftarrow 1$ **to** $u[s]$ **do** $CC_i^s \leftarrow C_i^s \oplus CCC_{i-1}^s$, $CCC_i^s \leftarrow CCC_i^r$
for $i \leftarrow u[s]+1$ **to** m **do**
    $CC_i^s \leftarrow C_i^s \oplus CCC_{i-1}^s$
    $CCC_i^s \xleftarrow{\$} \{0,1\}^n$; **if** $CCC_i^s \in$ Domain **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $CCC_i^s \xleftarrow{\$} \overline{\text{Domain}}$
    if $CC_i^s \in$ Range **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $CCC_i^s \leftarrow \pi^{-1}(CC_i^s)$
    $\pi(CCC_i^s) \leftarrow CC_i^s$, Domain $\leftarrow$ Domain $\cup \{CCC_i^s\}$, Range $\leftarrow$ Range $\cup \{CC_i^s\}$
$M^s \leftarrow 2\,(CCC_1^s \oplus CCC_m^s)$; **for** $i \in [1\,..\,m]$ **do** $PPP_i^s \leftarrow CCC_{m+1-i}^s \oplus M^s$
for $i \leftarrow 1$ **to** m **do**
    $PP_i^s \xleftarrow{\$} \{0,1\}^n$; **if** $PP_i^s \in$ Domain **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $PP_i^s \xleftarrow{\$} \overline{\text{Domain}}$
    if $PPP_i^s \in$ Range **then** $\mathsf{bad} \leftarrow \mathsf{true}$, $PP_i^s \leftarrow \pi^{-1}(PPP_i^s)$
    $P_i^s \leftarrow PP_i^s \oplus PPP_{i-1}^s$
    $\pi(PP_i^s) \leftarrow PPP_i^s$, Domain $\leftarrow$ Domain $\cup \{PP_i^s\}$, Range $\leftarrow$ Range $\cup \{PPP_i^s\}$
return $P_1 \cdots P_m$

Figure 3: A pseudo-code descibing the interaction of an attacker with the CMC mode. The highlighted statements are only executed after the flag bad is set.

Transforming the code. The user starts by adding a bad-event flag to the code of the CMC game (which in this case is set whenever there is an "accidental collision" between blocks that were supposed to be independent.) The tool can highlight all the statements that are only executed after a bad-event flag is set. The resulting game is called "CMC1" and its code is described in Figure 3. (This figure does not describe the main loop, but only the procedures that answer the adversary's queries. Also, this code assumes that there is no "tweak" $\mathbf{T}^*$, since this tweak is handled by a different lemma in the paper.)

The user deletes all the highlighted statements (which is a permissible transformation due to the "after-bad-is-set-nothing-matters" rule), and the resulting game is called "RND1". The proof then goes through a few sets of simple code transformations. In the first set, for example, the following changes are made:

I The permutation π is dropped from the code since it is never used anywhere. (The only things that are used are the sets Domain and Range, so the sets themselves are kept.)

II Instead of setting $CC_i^s \xleftarrow{\$} \{0,1\}^n$ and $C_i^s \leftarrow CC_i^s \oplus CCC_{i-1}^s$ (in lines 131 and 133), set $C_i^s \xleftarrow{\$} \{0,1\}^n$ and $CC_i^s \leftarrow C_i^s \oplus CCC_{i-1}^s$. Similar changes are made in lines 231 and 233.

Note that the first change, eliminating unused variables, is something that optimizing compilers easily do by themselves. As for the second change, the tool needs to have a rule in its "library of transformations", specifying that this transformation is permissible for variables of type "n-bit string".

Note that here one can already gain some things by having the automated tool. For one thing, it can do the π-elimination for us. But more importantly, the fact that the tool is doing the transformation saves one the trouble of (a) printing out the new code after the transformation and (b) verifying that this is indeed the code that you get by applying the transformations from above to the code from Figure 3. This code verification is non-trivial for a person to perform (what with the indexing and loops and everything) and in some cases it can lead to errors. Also, having to verify such mundane transformations is likely to deter many people from reading the proof.

The next set of modification to the code consists of moving some pieces of code around. Specifically, note that the variables C_i^s that are returned to A on encryption are now chosen uniformly at random, independently of other variables (cf. the transformation II from above). Hence, these variable can be returned to A immediately. Their values are recorded, however, and later used to determine the value of the flag bad. Similar changes also apply to the variables P_i^s on decryption queries.

The automated tool would check that these fragments of code can indeed be moved as desired without effecting the outcome of the procedure. (Moving code without changing the outcome is something that optimizing compilers routinely do.) Again, the main benefit from having a tool would be that the user don't have to verify that the new code matches the old one. The pseudo-code after all these transformations is depicted in Figure 4. In the CMC proof that game is called "RND3". (Another minor change was to eliminate M^s by using variable-substitution, replacing it with $2\,(PPP_1^s \oplus PPP_m^s)$.)

Eliminating the output. Next, the user will transform also the game with random output to arrive at the same game "RAND3". This is trivially done, since the random-output game differs from "RND3" only in things that are done after the output is determined. Hence, the user

Respond to the s-th adversary query as follows:

An encipher query, $\mathsf{Enc}(P_1^s \cdots P_m^s)$:

010 $\mathsf{type}^s \leftarrow \mathsf{Enc}$; $C^s = C_1^s \cdots C_m^s \xleftarrow{\$} \{0,1\}^{nm}$; **return** C^s

A decipher query, $\mathsf{Dec}(C_1^s \cdots C_m^s)$:

020 $\mathsf{type}^s \leftarrow \mathsf{Dec}$; $P^s = P_1^s \cdots P_m^s \xleftarrow{\$} \{0,1\}^{nm}$; **return** P^s

Finalization:

050 Domain $\leftarrow$ Range $\leftarrow \emptyset$; $\mathsf{bad} \leftarrow \mathsf{false}$

051 **for** $s \leftarrow 1$ **to** q **do**

100   **if** $\mathsf{type}^s = \mathsf{Enc}$ **then**

110   Let $u[s]$ be the largest index s.t. $P_1^s \cdots P_{u[s]}^s = P_1^r \cdots P_{u[s]}^r$ for some $r < s$

111         $PPP_0^s \leftarrow CCC_0^s \leftarrow 0^n$; **for** $i \leftarrow 1$ **to** $u[s]$ **do** $PP_i^s \leftarrow P_i^s \oplus PPP_{i-1}^s$, $PPP_i^s \leftarrow PPP_i^r$

112   **for** $i \leftarrow u[s]+1$ **to** m **do**

113             $PP_i^s \leftarrow P_i^s \oplus PPP_{i-1}^s$; $PPP_i^s \xleftarrow{\$} \{0,1\}^n$

114   **if** $PP_i^s \in$ Domain **or** $PPP_i^s \in$ Range **then** $\mathsf{bad} \leftarrow \mathsf{true}$

115   Domain $\leftarrow$ Domain $\cup PP_i^s\}$; Range $\leftarrow$ Range $\cup \{PPP_i^s\}$

120   **for** $i \in [1 \mathinner{..} m]$ **do** $CCC_i^s \leftarrow PPP_{m+1-i}^s \oplus 2\,(PPP_1^s \oplus PPP_m^s)$

130   **for** $i \leftarrow 1$ **to** m **do**

131             $CC_i^s \leftarrow C_i^s \oplus CCC_{i-1}^s$

132   **if** $CCC_i^s \in$ Domain **or** $CC_i^s \in$ Range **then** $\mathsf{bad} \leftarrow \mathsf{true}$

133   Domain $\leftarrow$ Domain $\cup \{CCC_i^s\}$; Range $\leftarrow$ Range $\cup \{CC_i^s\}$

200   The case $\mathsf{type}^s = \mathsf{Dec}$ is treated symmetrically

Figure 4: The pseudo-code after a few transformations. This game is called "RND3".

only needs to insert "dead code" (and a bad flag) into the game, which is clearly a permissible transformation.

Once both the CMC game and the random-output game were transformed to the same game, it becomes permissible to eliminate the output of that game. (As I explained on Page 5 in Section 2.1, this is permissible since the analysis is only concerned with the difference between the outputs of the two games, and once these games are unified there could be no difference between their outputs.) This "output elimination" only removes the **output** statement at the very end of the main loop.

Coin fixing. The next transformation in where the interaction is eliminated. This is done simply by viewing the C_i^s's and P_i^s's from lines 10 and 20 in Figure 4 not as random variables that are chosen uniformly at random, but rather as *inputs that are chosen by an adversary*. Notice that now the adversary is providing all the inputs and it never sees any outputs. Hence the game becomes non-interactive and quantifying over all adversaries is the same as quantifying over all possible inputs. The new game is called "NON1" in [HR03]. This type of transformation is called "coin fixing" in [BR04, Section 4.2].

Differently from all the other transformations so far, coin fixing actually changes the probability distribution of the variables in the game (but intuitively it is allowed because "it can only help the adversary"). Still, I believe that it is possible to verify automatically that it is permissible for the current code. For example, Bellare and Rogaway formulated in a condition under which the coin-fixing transformation is permissible, and it is plausible that that condition can be automatically verified. Alternatively, I now sketch a condition that can clearly be verified automatically (but is perhaps less general than the Bellare-Rogaway condition). First, it must be the case that the only things that "we care about" in the game are the probability of bad events. (I.e., coin-fixing cannot be applied to games where "we care about" the output.) For games of this type, it is always permissible to take variables that are chosen at random and immediately given to the adversary, and instead let the adversary specify the values of these variables.

Returning to the CMC proof, there is a wrinkle here, in that the adversary is only allowed to specify C_i^s's and P_i^s's that do not include *immediate collisions*: An immediate collision is defined as $C_i^s = C_i^r$ for $r < s$ and s is an encipher query, or $P_i^s = P_i^r$ for $r < s$ and s is a decipher query. It is argued that since such "immediate collisions" only happens with a small probability ϵ in the game "RND3", then even if the adversary is restricted as above, it still holds that Pr[RND3 sets bad] $\leq$ Pr[NON1 sets bad] $+ \epsilon$.

This last argument does not appear to be in the realm of "static code analysis", so it is not a-priory clear how to automate it. However, I still believe that this can be dome as follows: First, instead of having just one bad-event flag, there will be two flags, one for immediate collisions and one for all the others. Moreover, the code is modified so that it never sets both flags. Specifically, it first checks if there are immediate collisions, and only if there are none then it checks for the other collisions. Then the tool can eliminate the immediate-collision flag, as described on Page 5 in Section 2.1. Namely, whenever a variable is assigned a value from a random choice (e.g., $X \xleftarrow{\$} \{0,1\}^n$), the tool tags that variable as "random in $\{0,1\}^n$". Assume further that there is another variable Y that does not depend on X in terms of control-flow of the program. (I.e., nothing in the assignment of Y depends on the assignment of X. This can happen, for example, if X is assigned value after Y.) Then the tool can deduce that the event $X = Y$ only happens with probability 2^{-n}.

Now, since all the immediate collisions are of this type, and the code that checks for them is executed (at most once per iteration) in a loop that repeats qm times, then the tool can detect automatically that the probability of setting the immediate-collision flag is at most $qm/2^n$. Then the tool can eliminate the immediate-collision flag, and record a penalty of $qm/2^n$. Only after the immediate-collision flag is eliminates the user will proceed to the coin-fixing transformation, and the result is the game "NON1" from the CMC proof.

Before concluding the game-substitution sequence, the CMC proof makes two more changes to the code. First, the flag bad is only set when they are collisions in Domain and not when there are collisions in Range, and it is argued that by symmetry, this change can only reduce the probability of bad being set by at most a factor of two. Then, all the variables that do not effect the flag bad are eliminated from the code. (In particular the set Range is eliminated.) The resulting game is called "NON2", and this is the non-interactive game that is analyzed in the rest of the proof. The symmetry argument is rather "semantic", and I do not expect the tool to be of any help there, but once the first change is made, the tool can itself perform the second transformation. To justify the first transformation, the human prover could ask the tool for a printout of the game (preferably in a LaTeX-friendly format) and make this argument on paper. In fact, from the tool's perspective the are two impermissible transformations: one from "NON1" to "NON2" (which is justified by the symmetry argument) and one from "NON2" to the empty game (which eliminates the flag bad and is justified by the case analysis in the second half of the CMC proof).

Verifying the arguments. The above description only shows how the tool is used to generate proofs such as the CMC proof. This helps the person writing the proof to verify that his/her proof is correct, but more help is needed in order for others to verify the same.

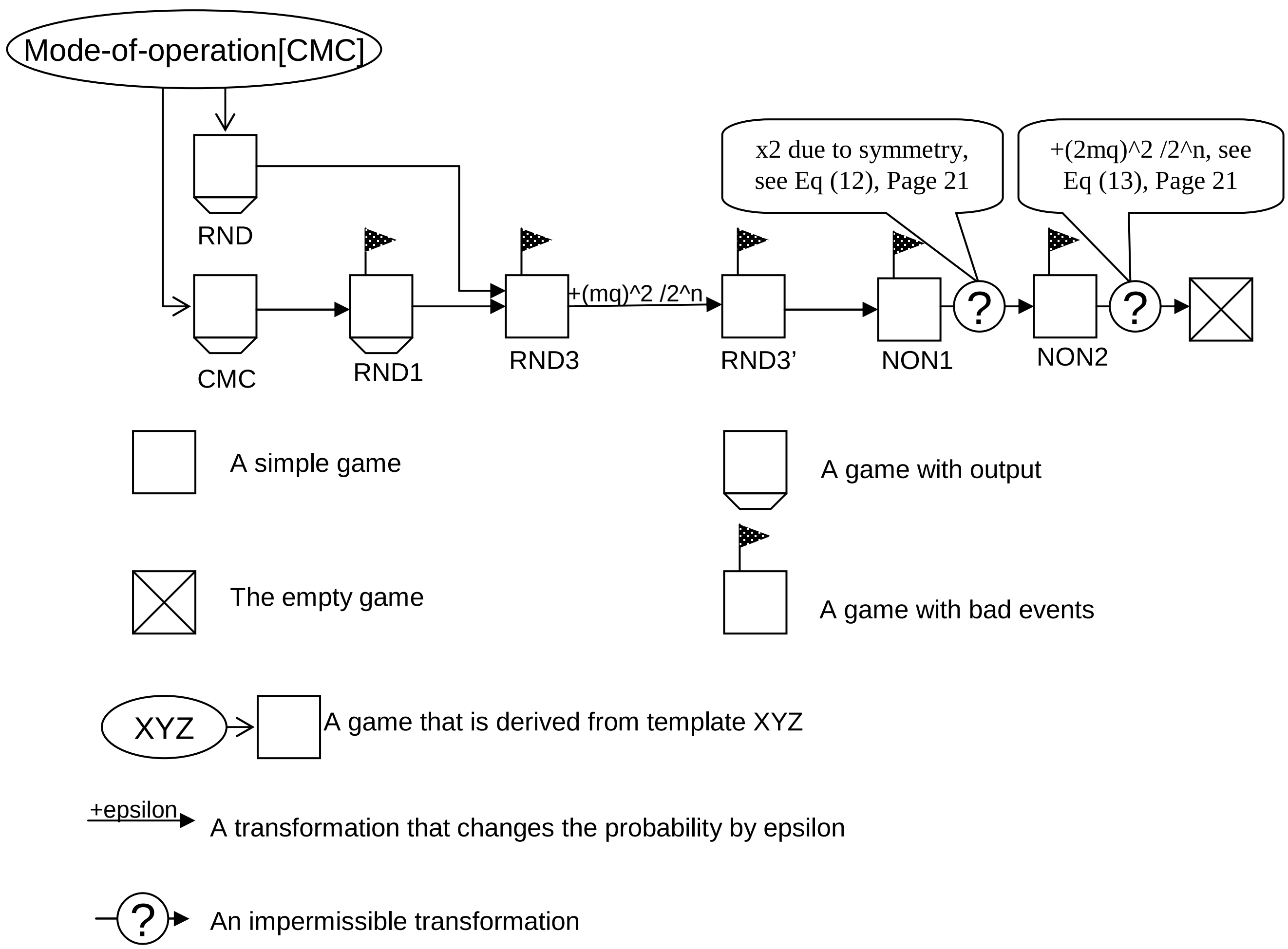

Figure 5: A pictorial description of the CMC proof of security

Essentially, I view the verification of the game-substitution argument as done simply by loading the tool and having it repeat the same process as was done during proof-generation. The tool can generate a pictorial representation of the proof that lets a human verifier see the overall structure of the proof, as demonstrated in Figure 5. The tool also stores the entire proof in a file. The human verifier can then load that file and ask the tool to show it what transformations happened from one step to the next.

Note that if one only cares about the truth of the CMC security theorem, then it is sufficient to verify just the two impermissible transformations at the end (since the tool verified everything else). A human will bother looking at the other transformations only if he/she thinks that the argument itself is interesting for some reason. Also, there is no longer any need to check for consistency of code. Namely, a human no longer needs to verify that "if I start from code A and make change X then I indeed get code B."

4 The Cramer-Shoup cryptosystem

The Cramer-Shoup public-key encryption scheme [CS01] uses some operations in an algebraic group $\mathbf{G}$ of prime order q, in conjunction with a hash function H that maps "long strings" into integers in the range $[0, q-1]$. Cramer and Shoup proved that this schemes resists adaptive chosen-ciphertext attacks, assuming that the decision-Diffie-Hellman problem is hard in the group $\mathbf{G}$, and the function H is target-collision-resistant (aka universal-one-way hash function [NY89]).

In more details, an instance of the Cramer-Shoup public-key encryption scheme is associated with an algebraic group $\mathbf{G}$ of a (known) prime order q, and a hash function H that takes a hashing key hk and a triple $(a, b, c) \in \mathbf{G}^3$ and outputs an integer in the range $[0, q-1]$, denoted $v \leftarrow H_{hk}(a, b, c)$. The key-generation, encryption, and decryption routines are defined as follows:

- The key-generation algorithm chooses two random elements $g, \hat{g} \in \mathbf{G} \setminus \{1\}$ and six random integers $x_1, x_2, y_1, y_2, z_1, z_2 \in [0, q-1]$, and a key hk for the hash function H. It computes $e \leftarrow g^{x_1}\hat{g}^{x_2}$, $f \leftarrow g^{y_1}\hat{g}^{y_2}$, and $h \leftarrow g^{z_1}\hat{g}^{z_2}$, and outputs the public key $(hk, g, \hat{g}, e, f, h)$ and the corresponding secret key $(hk, x_1, x_2, y_1, y_2, z_1, z_2)$.

- The encryption algorithm, on input the public key and a message $m \in \mathbf{G}$, chooses a random integer $u \in [0, q-1]$, computes $a \leftarrow g^u$, $\hat{a} \leftarrow \hat{g}^u$, $c \leftarrow h^u \cdot m$, $v \leftarrow H_{hk}(a, \hat{a}, c)$, and $d \leftarrow e^u f^{uv}$, and outputs the ciphertext $(a, \hat{a}, c, d)$.

- The decryption algorithm, on input the secret key and a ciphertext $(a, \hat{a}, c, d)$, tests that these are all elements of the group $\mathbf{G}$, computes $v \leftarrow H_{hk}(a, \hat{a}, c)$, and tests that $d = a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$. If all the tests pass it outputs the message $m \leftarrow c/a^{z_1}\hat{a}^{z_2}$ (and otherwise it outputs $\perp$).

A pseudo-code for these procedures is given in Figure 6 (and it should be easy to turn this pseudo-code into a "real code"). Note that this code contains group operations (such as multiplication and exponentiation) that in a real code would be implemented via calls to some library routines. Namely, the tool would have a library that includes routines for dealing with operations in a prime-order group. (See more discussion of these routines later in this section.)

Key-Generation$(\mathbf{G}, q, H)$: // $\mathbf{G}$ is a group of prime order q, and H is a
$g, \hat{g} \xleftarrow{\$} \mathbf{G} \setminus \{1\}$ // keyed hash function from $\mathbf{G}^3$ to $[0, q-1]$
$x_1, x_2, y_1, y_2, z_1, z_2 \xleftarrow{\$} [0, q-1]$
$e \leftarrow g^{x_1} \hat{g}^{x_2}, \;\; f \leftarrow g^{y_1} \hat{g}^{y_2}, \;\; h \leftarrow g^{z_1} \hat{g}^{z_2}$
$hk \xleftarrow{\$}$ key-space-for-the-hash-function-H
return $((hk, g, \hat{g}, e, f, h), \; (hk, x_1, x_2, y_1, y_2, z_1, z_2))$

Encryption$((\mathbf{G}, q, H), (hk, g, \hat{g}, e, f, h), m)$:
assert $m \in \mathbf{G}$ // Else return $\perp$
$u \xleftarrow{\$} [0, q-1]$
$a \leftarrow g^u, \;\; \hat{a} \leftarrow \hat{g}^u$
$b \leftarrow h^u, \;\; c \leftarrow b \cdot m$
$v \leftarrow H_{hk}(a, \hat{a}, c)$
$d \leftarrow e^u \cdot f^{uv}$
return $(a, \hat{a}, c, d)$

Decryption$((\mathbf{G}, q, H), (hk, g, \hat{g}, x_1, x_2, y_1, y_2, z_1, z_2), (a, \hat{a}, c, d))$:
assert $a, \hat{a}, c, d \in \mathbf{G}$ // Else return $\perp$
$v \leftarrow H_{hk}(a, \hat{a}, c)$
if $d \neq a^{x_1 + v y_1} \cdot \hat{a}^{x_2 + v y_2}$ **then return** $\perp$
$b \leftarrow a^{z_1} \hat{a}^{z_2}$
$m \leftarrow c \cdot b^{-1}$, **return** m

Figure 6: A pseudo-code descibing the Cramer-Shoup cryptosystem.

4.1 The proof of security

The proof of security in [CS01, Theorem 6.1] is already organized as a sequence of games. The first game describes an adversary A that mounts a chosen-ciphertext attack on an instance of the Cramer-Shoup cryptosystem. This game is called $\mathbf{CCA}_0$ (it was called $\mathbf{G}_0$ in [CS01]). A pseudo-code description the game $\mathbf{CCA}_0$ can be found in Figure 7. Note the following about that code:

- The only thing in the code in Figure 7 that is specific to the Cramer-Shoup scheme, is the definition of the system parameters and the message domain. Hence it should be easy to automatically generate the code of the CCA game from the code of the encryption scheme itself.

  It should be noted that it is more useful to think of the "function calls" to the key-generation, encryption, and decryption from the code in Figure 7 as if they were actually macro expansions. Namely, one should think of pasting the code of these procedures in the place when they are called.

- The calls to the decryption routine from within the code in Figure 7 have an additional parameter s, which is the query number. The encryption and decryption routines are exactly those from Figure 6, except that in the decryption routine, each internal variable in the code is replaced with a sequence of variables, one for each value of s, for example $a[s]$ instead of a, $v[s]$ instead of v, etc. (The encryption routine is only called once during the CCA game, so there is no need to add indexing to its variables.)

```
CCA-Game (parameters: a bit b_CCA and a bound t_CCA on the number of adversary queries):
params ← (G, q, H), Domain ← G                    // Parameters of this CS instance
(pk, sk) ← Key-Generation(params)

C[1] ← Adversary(params, pk)                      // First phase: t_CCA decryption queries
for s ← 1 to t_CCA do
      answer[s] ← Decryption(s; params, sk, C[s])
      if s < t_CCA then C[s + 1] ← Adversary(answer[s])

(M*_0, M*_1) ← Adversary(answer[t_CCA])           // The target plaintext
assert M*_0, M*_1 ∈ Domain                        // Else output 0
C* ← Encryption(params, pk, M*_{b_CCA})

C[t_CCA + 1] ← Adversary(C*)                      // Second phase: t_CCA more decryption queries
for s ← t_CCA + 1 to 2t_CCA do
      assert C[s] ≠ C*                            // Else output 0
      answer[s] ← Decryption(s; params, sk, C[s])
      if s < 2t_CCA then C[s + 1] ← Adversary(answer[s])

guess ← Adversary(answer[2t_CCA])                 // The adversary's guess
if guess = 1 then output 1, else output 0
```

Figure 7: Pseudo-code for a chosen-ciphertext attack.

- The game $\mathbf{CCA}_0$ is parametrized by the bit b_{CCA} (that determines which of the two messages to encrypt in an Enc query). The goal in the analysis is to bound the difference between the probability of outputting '1' when $b_{CCA} = 1$ and outputting '1' when $b_{CCA} = 0$.

  The game is also parametrized by a bound t_{CCA} on the number of decryption queries that the adversary makes. For convenience, the code assumes that the adversary makes exactly t_{CCA} decryption queries in each of the two phases of the game. (This is convenient since it is easier to deal with loops that repeat a fixed number of times than with loops that repeat a variable number of times.)

Game $\mathbf{CCA}_1$. The first step in the analysis is to modify the encryption procedure so that it uses the secret key instead of the public key, but in a way that preserve the distribution of all the variables in the game. In terms of the code from Figures 6 and 7, the statements $b \leftarrow h^u$ and $d \leftarrow e^u \cdot f^{uv}$ (lines 114 and 116) are replaced with $b \leftarrow a^{z_1}\hat{a}^{z_2}$ and $d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$. It is quite easy to prove on paper that this transformation preserves the values of all the variables in the CCA game. For example, one can check that

$$h^u \overset{(a)}{=} (g^{z_1}\hat{g}^{z_2})^u \overset{(b)}{=} (g^u)^{z_1}(\hat{g}^u)^{z_2} \overset{(c)}{=} a^{z_1}\hat{a}^{z_2}$$

where equality (a) is due to the assignment of h in line 103, equality (b) is due to the laws of multiplication and exponentiation, and equality (c) is due to the assignments of $a, \hat{a}$ in line 113.

Verifying these conditions with an automated tool seems a bit harder, though. The user can make the change to the code in three steps, corresponding to the equalities (a)–(c) above, and the tool could automatically verify that each of these small steps is permissible. Verifying steps (a) and (c) is easy, since this is just variable substitution. Verifying step (b), on the other hand, requires that the tool knows about the laws of multiplication and exponentiation. This knowledge is not different in principle from the knowledge that is needed to determine that the code $x \xleftarrow{\$} \{0,1\}^n$, $y \leftarrow x \oplus c$ is

Encryption$((\mathbf{G}, q, H), (hk, g, \hat{g}, x_1, x_2, y_1, y_2, z_1, z_2), m)$:
111 **assert** $m \in \mathbf{G}$ // Else return $\perp$
112 $u, u' \xleftarrow{\$} [0, q-1]$
113 $a \leftarrow g^u, \;\; \hat{a} \leftarrow \hat{g}^{u'}$
114 $b \leftarrow a^{z_1} \hat{a}^{z_2}, \;\; c \leftarrow b \cdot m$
115 $v \leftarrow H_{hk}(a, \hat{a}, c)$
116 $d \leftarrow a^{x_1 + vy_1} \cdot \hat{a}^{x_2 + vy_2}$
117 **return** $(a, \hat{a}, c, d)$

Figure 8: The modified encryption routine in game $\mathbf{CCA}_2$.

equivalent to the code $y \xleftarrow{\$} \{0,1\}^n$, $x \leftarrow y \oplus c$. Such knowledge can be built into the tool together with the library routines that specify the operations in a prime-order group.

Game $\mathbf{CCA}_2$. The next modification is where the proof relies on the hardness of decision-Diffie-Hellman (DDH) in the group $\mathbf{G}$. That is, it is assumed that given four elements $(g, \hat{g}, a, \hat{a})$ in $\mathbf{G}$, it is infeasible to determine whether they were chosen as four independent random elements (with $g, \hat{g} \neq 1$), or they were chosen at random subject to the condition $DL_g(a) = DL_{\hat{g}}(\hat{a})$ (where DL is the discrete-logarithm function in the group $\mathbf{G}$).

The game $\mathbf{CCA}_1$ is modified by replacing the assignment $\hat{a} \leftarrow \hat{g}^u$ in line 113 by $\hat{a} \leftarrow \hat{g}^{u'}$, where u' is a new variable that is chosen at random in line 112, $u' \xleftarrow{\$} [0, q-1]$. That is, instead of having $DL_g(a) = DL_{\hat{g}}(\hat{a})$, now $DL_g(a)$ and $DL_{\hat{g}}(\hat{a})$ are two independent variables, each uniform in $[0, q-1]$. The code of the modified encryption routine is given in Figure 8, and the new game is called $\mathbf{CCA}_2$. Clearly, the games $\mathbf{CCA}_1$ and $\mathbf{CCA}_2$ induce different distribution on their common variables. However, the analysis would prove that these two distributions cannot be distinguished by A, based on the hardness of DDH problem in the group $\mathbf{G}$. This is done by describing yet more games, constituting *a reduction* from distinguishing between $\mathbf{CCA}_1$ and $\mathbf{CCA}_2$ to breaking DDH.

Specifically, two new games are described, one that includes a distinguisher D with input $(g, \hat{g}, g^u, \hat{g}^u)$, and another that includes the same D but with input $(g, \hat{g}, g^u, \hat{g}^{u'})$ (where u, u' are independently uniform in $[0, q-1]$). Then the user shows that the latter game induces the same probability distribution over its variables as $\mathbf{CCA}_2$, while the former induces the same probability distribution as $\mathbf{CCA}_1$. It thus follows that distinguishing between $\mathbf{CCA}_1$ and $\mathbf{CCA}_2$ is as hard as solving DDH, and in particular $\mathbf{CCA}_2$ outputs one with the same probability as $\mathbf{CCA}_1$ (up to a negligible difference).

One plausible way of automating this argument is as follows. The tool could have a template for "reduction to DDH", and the user specifies that the transformation from $\mathbf{CCA}_1$ to $\mathbf{CCA}_2$ should be justified using that template. The template is then instantiated, with the group $\mathbf{G}$ and its order q as parameters. This generates a code template for a distinguisher D and its input, as shown in Figure 9. (This code actually describes just one game, with a parameter $b_{DDH} \in \{0,1\}$ that determines how the input to D is generated.) The user then needs to fill in the code for the distinguisher D by morphing the code of the games $\mathbf{CCA}_1$ and $\mathbf{CCA}_2$.

Once this is done, the tool would verify that indeed the "code transformation" from $\mathbf{CCA}_1$ to the DDH game with $b_{DDH} = 1$ is permissible, and similarly for the transformation from the DDH game with $b_{DDH} = 0$ to $\mathbf{CCA}_2$. The tool may either be able to verify this in one shot, or the user will have to take it step-by-step via a sequence of little permissible transformations that lead from

DDH-Game (parameter: a bit $\mathsf{b}_{\mathsf{DDH}}$)

$\mathsf{params} \leftarrow (\mathbf{G}, q)$ // Parameters of this DDH group
$g, \hat{g} \xleftarrow{\$} \mathbf{G} \setminus \{1\},\ \ u, u' \xleftarrow{\$} [0, q-1]$
if $\mathsf{b}_{\mathsf{DDH}} = 0$ **then** $a \leftarrow g^u$, $\hat{a} \leftarrow \hat{g}^{u'}$
else $a \leftarrow g^u$, $\hat{a} \leftarrow \hat{g}^u$

$\mathsf{guess} \leftarrow D(\mathsf{params}, g, \hat{g}, a, \hat{a})$
if $\mathsf{guess} = 1$ **then output** 1, **else output** 0

Figure 9: A pseudo-code template for DDH reduction.

one code to the other. (In this specific example of the Cramer-Shoup reduction, almost all the transformations are just code movements.)

Some comments. I believe that "reduction steps" are where an automated tool will be most helpful. First, the tool helps the user keep track of what is the current code and how to transform it next, which may be hard to do if there is a sequence of more than three or four transformations in the proof. Even more importantly, the tool can let the user verify that the code from the reduction indeed complies with the assumption that are made, which may be hard to check in some cases. (Indeed, there are instances of proofs in the literature that fail for exactly that reason.)

As an example for the latter point, think of trying to do the reduction to DDH directly from the code of the original game $\mathbf{CCA}_0$, rather than doing first the transformation to $\mathbf{CCA}_1$. Denote by $\mathbf{CCA}'_2$ the game that is obtained by modifying game $\mathbf{CCA}_0$, changing $\hat{a} \leftarrow \hat{g}^u$ to $\hat{a} \leftarrow \hat{g}^{u'}$ in line 113. One may attempt to reduce distinguishing $\mathbf{CCA}_0$ from $\mathbf{CCA}'_2$ to the DDH problem in $\mathbf{G}$, but notice that now the value u is used in computing b and d (cf. lines 114 and 116 in Figure 6). Hence, trying to do a reduction to DDH will result in a distinguisher $D(g, \hat{g}, a, \hat{a})$ *that uses the discrete-logarithm of a base g for its internal computations*, and the user will not be able to find a sequence of permissible transformations between the code that was obtained from the template and the code of $\mathbf{CCA}'_2$.

The example of the Cramer-Shoup proof may not sufficiently illustrate the complexity of a reduction step (or the usefulness of an automated help). This is because there is only a reduction to a very simple "non-interactive assumption" (i.e., DDH), and it occurs very close to the beginning of the game sequence. In general, however, one may need to use a reduction to a complex "interactive assumption" that only happen after five or more transformations. (For example, think of a reduction from distinguishing between game-6 and game-7 in a sequence to, say, the problem of simulation-sound zero-knowledge.)

Finally, note that in some cases there may be a more complex setting of "recursive reductions". For example, instead of having only permissible transformation between the game $\mathbf{CCA}_1$ and the DDH game with $b_{DDH} = 1$, some proofs may need to use another "reduction step" between these two, with a reduction to another hard problem.

Game $\mathbf{CCA}_3$. The next transformation modifies the decryption routine, so that it rejects the ciphertext whenever $DL_g(a) \neq DL_{\hat{g}}(\hat{a})$. This is done by first modifying the way $\hat{g}$ is chosen, replacing the choices $g, \hat{g} \xleftarrow{\$} \mathbf{G} \setminus \{1\}$ (line 101 in Figure 6) by $g \xleftarrow{\$} \mathbf{G} \setminus \{1\}$, $w \xleftarrow{\$} [1, q-1]$ and $\hat{g} \leftarrow g^w$. (The tool should have a rule saying that this transformation is always permissible in a

Decryption$((\mathbf{G}, q, H), (hk, g, w, x_1, x_2, y_1, y_2, z_1, z_2), (a[s], \hat{a}[s], c[s], d[s]))$:
121 **assert** $a[s], \hat{a}[s], c[s], d[s] \in \mathbf{G}$ // Else return $\perp$
122 $v[s] \leftarrow H_{hk}(a[s], \hat{a}[s], c[s])$
123 **if** $d[s] = a[s]^{x_1+v[s]y_1} \cdot \hat{a}[s]^{x_2+v[s]y_2}$ **and** $\hat{a}[s] \neq a[s]^w$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$, **return** $\perp$
123a **if** $d[s] \neq a[s]^{x_1+v[s]y_1} \cdot \hat{a}[s]^{x_2+v[s]y_2}$ **then return** $\perp$
124 $b[s] \leftarrow a[s]^{z_1} \hat{a}[s]^{z_2}$
125 $m[s] \leftarrow c[s] \cdot b[s]^{-1}$, **return** $m[s]$

Figure 10: The modified decryption routine in $\mathbf{CCA}_3$.

prime order group).

Then the decryption routine verifies that $\hat{a}[s] = a[s]^w$ (and outputs $\perp$ otherwise). Moreover, a bad-event flag bad_1 is added to the code (for purpose of future analysis), and this flag is set whenever some ciphertext $(a[s], \hat{a}[s], c[s], d[s])$ would have passed the test of the encryption routine in $\mathbf{CCA}_2$ but fail the new test $\hat{a}[s] = a[s]^w$ in $\mathbf{CCA}_3$. A pseudo-code of the modified decryption routine can be found in Figure 10. Note that the game $\mathbf{CCA}_3$ is identical to $\mathbf{CCA}_2$ except for what happens after the flag bad_1 is set. This is a permissible transformation, as was discussed in Section 2.1.

Next, the setting of bad_1 is separated from returning $\perp$, replacing lines 123 and 123a with the equivalent lines:

123 **if** $d[s] = a[s]^{x_1+v[s]y_1} \cdot \hat{a}[s]^{x_2+v[s]y_2}$ **and** $\hat{a}[s] \neq a[s]^w$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$
123a **if** $\hat{a}[s] \neq a[s]^w$ **or** $d[s] \neq a[s]^{x_1+v[s]y_1} \cdot \hat{a}[s]^{x_2+v[s]y_2}$ **then return** $\perp$

Next, the check $d[s] \stackrel{?}{=} a[s]^{x_1+vy_1} \cdot \hat{a}[s]^{x_2+vy_2}$ in line 123a is replaced by $d[s] \stackrel{?}{=} a[s]^{(x_1+wx_2)+v(y_1+wy_2)}$, and similarly the assignment $b[s] \leftarrow a[s]^{z_1}\hat{a}[s]^{z_2}$ in line 124 is replaced by $b[s] \leftarrow a[s]^{z_1+wz_2}$. Note that since these statements are only executed if $\hat{a}[s] = a[s]^w$, then this code is still equivalent to the previous one (and automatically verifying this equivalence pose no more difficulties that were not already encountered in previous steps). Also, the assignments of e, f, h during the key generation (line 103) are replaced by $e = g^{x_1+wx_2}$, $f = g^{y_1+wy_2}$ and $g = f^{z_1+wz_2}$. Now set $x \leftarrow x_1 + wx_2$, $y \leftarrow y_1 + wy_2$, and $z \leftarrow z_1 + wz_2$, and use x, y, z in line 103 of the key-generation and lines lines 123a and 124 of the decryption routine.

Finally, the setting of the flag bad_1 is delayed until the end of the game. (This is just code movement, and the tool should be able to verify that it is permissible.) Moreover, note that $x_1, x_2, y_1, y_2 z_1, z_2$ are no longer used in the decryption routine, so their setting is deferred until the first time they are used (i.e., during the encryption query). The resulting game is called $\mathbf{CCA}'_3$ (and it is roughly equivalent to game $\mathbf{G}_3$ from [CS01]). It should be possible to automatically verify that all these transformations are permissible (they do not raise any new issues that were not dealt with in previous steps). The complete pseudo-code of the $\mathbf{CCA}'_3$ is found in Figure 11. In Figure 11 I actually pasted the key-generation, encryption, and decryption code in the place that the previous code had calls to these routines.)

Game $\mathbf{CCA}_4$. The next transformation eliminates the dependence of the game on the parameter b_{CCA}. Namely, the assignment $c \leftarrow a^{z_1}\hat{a}^{z_2} \cdot m_{b_{CCA}}$ (line 025 in the encryption of the target plaintext) is replaced by simply $c \leftarrow g^r$ for a random $r \stackrel{\$}{\leftarrow} [0, q-1]$. Justifying this transformation involves

$\mathbf{CCA}_3'$ Game (parameters: a bit b_{CCA} and a bound t_{CCA} on the number of adversary queries):

$\mathsf{params} \leftarrow (\mathbf{G}, q, H)$, Domain $\leftarrow \mathbf{G}$ // Parameters of this CS instance
$g \xleftarrow{\$} \mathbf{G} \setminus \{1\}$, $w \xleftarrow{\$} [1, q-1]$, $\hat{g} \leftarrow g^w$ // Key generation
$x, y, z, \xleftarrow{\$} [0, q-1]$
$e \leftarrow g^x,\ \ f \leftarrow g^y,\ \ h \leftarrow g^z$
$hk \xleftarrow{\$}$ key-space-for-the-hash-function-H

$(a[1], \hat{a}[1], c[1], d[1]) \leftarrow$ **Adversary**$(\mathsf{params}, (hk, g, \hat{g}, e, f, h))$
for $s \leftarrow 1$ **to** t_{CCA} **do**
  $v[s] \leftarrow H_{hk}(a[s], \hat{a}[s], c[s])$
  if $a[s], \hat{a}[s], c[s], d[s] \in \mathbf{G}$ **and** $\hat{a}[s] = a[s]^w$ **and** $d[s] = a[s]^{x+vy}$
    then $b[s] \leftarrow a[s]^z,\ \ m[s] \leftarrow c[s] \cdot b[s]^{-1}$
    else $m[s] \leftarrow \perp$
  if $s < \mathsf{t}_{\text{CCA}}$ **then** $(a[s+1], \hat{a}[s+1], c[s+1], d[s+1]) \leftarrow$ **Adversary**$(m[s])$

$(m_0, m_1) \leftarrow$ **Adversary**$(m[\mathsf{t}_{\text{CCA}}])$ // Encrypt the target plaintext
assert $m_0, m_1 \in$ Domain // Else output 0
$u, u' \xleftarrow{\$} [0, q-1]$
$a \leftarrow g^u,\ \ \hat{a} \leftarrow \hat{g}^{u'}$
$z_2 \xleftarrow{\$} [0, q-1],\ \ z_1 \leftarrow z - wz_2 \bmod q$
$c \leftarrow a^{z_1} \hat{a}^{z_2} \cdot m_{b_{CCA}}$
$v \leftarrow H_{hk}(a, \hat{a}, c)$
$x_2, y_2 \xleftarrow{\$} [0, q-1],\ \ x_1 \leftarrow x - wx_2 \bmod q,\ \ y_1 \leftarrow y - wy_2 \bmod q$
$d \leftarrow a^{x_1 + vy_1} \cdot \hat{a}^{x_2 + vy_2}$

$(a[\mathsf{t}_{\text{CCA}}+1], \hat{a}[\mathsf{t}_{\text{CCA}}+1], c[\mathsf{t}_{\text{CCA}}+1], d[\mathsf{t}_{\text{CCA}}+1]) \leftarrow$ **Adversary**$(a, \hat{a}, c, d)$
for $s \leftarrow \mathsf{t}_{\text{CCA}} + 1$ **to** $2\mathsf{t}_{\text{CCA}}$ **do**
  assert $(a[s], \hat{a}[s], c[s], d[s]) \neq (a, \hat{a}, c, d)$ // Else output 0
  $v[s] \leftarrow H_{hk}(a[s], \hat{a}[s], c[s])$
  if $a[s], \hat{a}[s], c[s], d[s] \in \mathbf{G}$ **and** $\hat{a}[s] = a[s]^w$ **and** $d[s] = a[s]^{x+vy}$
    then $b[s] \leftarrow a[s]^z,\ \ m[s] \leftarrow c[s] \cdot b[s]^{-1}$
    else $m[s] \leftarrow \perp$
  if $s < 2\mathsf{t}_{\text{CCA}}$ **then** $(a[s+1], \hat{a}[s+1], c[s+1], d[s+1]) \leftarrow$ **Adversary**$(m[s])$

$\mathsf{guess} \leftarrow$ **Adversary**$(m[2\mathsf{t}_{\text{CCA}}])$ // The adversary's guess
$\mathsf{bad}_1 \leftarrow \mathsf{false}$ //Compute the flag bad_1 before exiting
for $s \leftarrow 1$ **to** $2\mathsf{t}_{\text{CCA}}$ **do**
  if $d[s] = a[s]^{x_1 + v[s]y_1} \cdot \hat{a}[s]^{x_2 + v[s]y_2}$ **and** $\hat{a}[s] \neq a[s]^w$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$
if $\mathsf{guess} = 1$ **then output** 1, **else output** 0

Figure 11: Pseudo-code for the game $\mathbf{CCA}_3'$.

some algebraic manipulations, and perhaps it is simpler to do these on paper than to try and use the tool for them. Still, I describe below a sequence of steps that can be done in order to have the tool verify that this transformation is permissible.

First, the tool should have also the discrete-logarithm function that takes as input $g, h \in \mathbf{G}$ and returns $DL_g(h)$. It should be stressed that from the tool's perspective, the code of all these games is just a static object that needs never be executed, so the fact that discrete-logarithm cannot be efficiently computed should not prevent the tool from having it in the code.[2] However, this function must be tagged as "non-computable" which means that although is can be used in an intermediate game in the sequence, it should never be used in a game that should be executable. In particular, it cannot be used if the current code transformation is a "reduction step". Also, the tool should verify that $DL_g(h)$ actually exists, by verifying that $g, h \in \mathbf{G}$ and that g is either an element that cannot possibly be equal to one (as in the code, **if** $g \neq 1$ **then** $u \leftarrow DL_g(h)$), or a random element[3] in $\mathbf{G}$.

Then, the assignment $c \leftarrow a^{z_1}\hat{a}^{z_2} \cdot m_{b_{CCA}}$ is first replaced by $c \leftarrow g^{u(z-wz_2)} \cdot g^{wu'z_2} \cdot g^{DL_g(m_{b_{CCA}})}$, and then replaced again by setting $r \leftarrow uz + (u' - u)wz_2 + DL_g(m_{b_{CCA}}) \bmod q$ and $c \leftarrow g^r$. Next, the assignment for z_2 and r are replaced by $r \xleftarrow{\$} [0, q-1]$ and $z_2 \leftarrow (r - uz - DL_g(m_{b_{CCA}}))/(u' - u)$ (with "penalty" of $1/q$ to account for the case $u = u'$). Finally, z_1, z_2 are eliminated from the code altogether, since they are no longer used anywhere (and in the process also the use of the function $DL_g(m)$ is eliminated).

Now the code no longer depends on the parameter b_{CCA} so this parameter can be removed. More importantly, eliminating the output of the game becomes a permissible transformation. All that is left is a game with one bad-event flag, and the analysis needs to bound the probability of it being set.

Game $\mathbf{CCA}_5$. This step finally makes use of the target-collision-resistance (TCR) of the hash function H. First, since the encryption phase no longer depends on the target plaintext messages, it can be moved to the beginning of the game. Also, the assertion $(a[s], \hat{a}[s], c[s], d[s]) \neq (a, \hat{a}, c, d)$ is added to the decryption queries in the first phase. Since $a, \hat{a}, c, d$ are chosen independently of the adversary's view, then this alters the distribution by at most $\mathtt{t}_{CCA}/q^4$. I stress that although this sounds like a "semantic argument", it should be possible to use static code analysis to derive this bound: An static program-flow analysis reveals that indeed $(a[s], \hat{a}[s], c[s], d[s])$ are independent of $(a, \hat{a}, c, d)$ (this is the same analysis that allows to move the choice of $(a, \hat{a}, c, d)$ before the choice of $(a[s], \hat{a}[s], c[s], d[s])$). Moreover, the tool can be endowed with a rule asserting that a statement such as $u \xleftarrow{\$} [0, q-1], \quad x \leftarrow g^u$ implies that whenever x is independent of a (in the sense of a program-flow) then $a = x$ only holds with probability at most $1/q$. Hence the tool should be able to infer that since the assertion appears inside a loop that repeats $\mathtt{t}_{CCA}$ times and it includes a conjunction of four of these $1/q$ probability (all independent in the sense of program-flow), then adding it involves a "penalty" of at most $\mathtt{t}_{CCA}/q^4$.

Then another flag bad_2 is added, which is set whenever there is an index s such that $v[s] = v$ but $(a[s], \hat{a}[s], c[s]) \neq (a, \hat{a}, c)$. If this happens then the corresponding decryption returns $\perp$. Notice that this transformation is permissible, as it is just another standard use of a bad-event flag. The code of $\mathbf{CCA}_5$ is described in Figure 12.

[2] This is how we represent in the tool an argument that begins with "let $\alpha = DL_g(h)$...".

[3] If g is random then computing $DL_g(h)$ involves a "penalty" of $1/q$, to account for the possibility that $g = 1$.

$\mathbf{CCA}_5$ Game (parameters: a bound $\mathsf{t}_{\mathsf{CCA}}$ on the number of adversary queries):

$\mathsf{params} \leftarrow (\mathbf{G}, q, H)$, Domain $\leftarrow \mathbf{G}$ // Parameters of this CS instance
$\mathsf{bad}_1 \leftarrow \mathsf{bad}_2 \leftarrow \mathsf{bad}_3 \leftarrow \mathsf{false}$
$g \xleftarrow{\$} \mathbf{G} \setminus \{1\}$, $w \xleftarrow{\$} [1, q-1]$, $\hat{g} \leftarrow g^w$
$x, y, z, \xleftarrow{\$} [0, q-1]$, $e \leftarrow g^x$, $f \leftarrow g^y$, $h \leftarrow g^z$
$u, u' \xleftarrow{\$} [0, q-1]$, $a \leftarrow g^u$, $\hat{a} \leftarrow \hat{g}^{u'}$
$r \xleftarrow{\$} [0, q-1]$, $c \leftarrow g^r$
$hk \xleftarrow{\$}$ key-space-for-the-hash-function-H
$v \leftarrow H_{hk}(a, \hat{a}, c)$
$x_2, y_2 \xleftarrow{\$} [0, q-1]$, $x_1 \leftarrow x - wx_2 \bmod q$, $y_1 \leftarrow y - wy_2 \bmod q$
$d \leftarrow a^{x_1+vy_1} \cdot \hat{a}^{x_2+vy_2}$

$(a[1], \hat{a}[1], c[1], d[1]) \leftarrow \mathbf{Adversary}(\mathsf{params}, (hk, g, \hat{g}, e, f, h))$
for $s \leftarrow 1$ **to** $\mathsf{t}_{\mathsf{CCA}}$ **do**
  assert $(a[s], \hat{a}[s], c[s], d[s]) \neq (a, \hat{a}, c, d)$ // Else output 0
  $v[s] \leftarrow H_{hk}(a[s], \hat{a}[s], c[s])$
  if $a[s], \hat{a}[s], c[s], d[s] \in \mathbf{G}$ **and** $\hat{a}[s] = a[s]^w$ **and** $d[s] = a[s]^{x+vy}$ **then**
    $b[s] \leftarrow a[s]^z$, $m[s] \leftarrow c[s] \cdot b[s]^{-1}$
  else $m[s] \leftarrow \perp$
  if $v[s] = v$ **then** $\mathsf{bad}_3 \leftarrow \mathsf{true}$, $m[s] \leftarrow \perp$
  if $s < \mathsf{t}_{\mathsf{CCA}}$ **then** $(a[s+1], \hat{a}[s+1], c[s+1], d[s+1]) \leftarrow \mathbf{Adversary}(m[s])$

$(m_0, m_1) \leftarrow \mathbf{Adversary}(m[\mathsf{t}_{\mathsf{CCA}}])$
assert $m_0, m_1 \in$ Domain // Else output 0

$(a[\mathsf{t}_{\mathsf{CCA}}+1], \hat{a}[\mathsf{t}_{\mathsf{CCA}}+1], c[\mathsf{t}_{\mathsf{CCA}}+1], d[\mathsf{t}_{\mathsf{CCA}}+1]) \leftarrow \mathbf{Adversary}(a, \hat{a}, c, d)$
for $s \leftarrow \mathsf{t}_{\mathsf{CCA}}+1$ **to** $2\mathsf{t}_{\mathsf{CCA}}$ **do**
  assert $(a[s], \hat{a}[s], c[s], d[s]) \neq (a, \hat{a}, c, d)$ // Else output 0
  $v[s] \leftarrow H_{hk}(a[s], \hat{a}[s], c[s])$
  if $a[s], \hat{a}[s], c[s], d[s] \in \mathbf{G}$ **and** $\hat{a}[s] = a[s]^w$ **and** $d[s] = a[s]^{x+vy}$ **then**
    $b[s] \leftarrow a[s]^z$, $m[s] \leftarrow c[s] \cdot b[s]^{-1}$
  else $m[s] \leftarrow \perp$
  if $v[s] = v$ **then** $\mathsf{bad}_3 \leftarrow \mathsf{true}$, $m[s] \leftarrow \perp$
  if $s < 2\mathsf{t}_{\mathsf{CCA}}$ **then** $(a[s+1], \hat{a}[s+1], c[s+1], d[s+1]) \leftarrow \mathbf{Adversary}(m[s])$

$\mathsf{guess} \leftarrow \mathbf{Adversary}(m[2\mathsf{t}_{\mathsf{CCA}}])$ // The adversary's guess
if $u = u'$ **then** $\mathsf{bad}_2 \leftarrow \mathsf{true}$
for $s \leftarrow 1$ **to** $2\mathsf{t}_{\mathsf{CCA}}$ **do**
  if $d[s] = a[s]^{x_1+v[s]y_1} \cdot \hat{a}[s]^{x_2+v[s]y_2}$ **and** $\hat{a}[s] \neq a[s]^w$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$
if $\mathsf{guess} = 1$ **then output** 1, **else output** 0

Figure 12: Pseudo-code for the game $\mathbf{CCA}_5$.

TCR-Game (parameter: a Hash function H)
000 $\mathsf{bad} \leftarrow \mathsf{false}$
001 $m_0 \leftarrow CF(H)$ // CF is a collision-finder
002 $hk \xleftarrow{\$}$ key-space-for-the-hash-function-H
003 $m_1 \leftarrow CF(hk)$
004 **if** $H_{hk}(m_0) = H_{hk}(m_1)$ **then** $\mathsf{bad} \leftarrow \mathsf{true}$

Figure 13: The code template for target-collision-resistant hashing

The user now do another reduction, proving that the flag bad_2 is rarely set because the hash function H is target-collision-resistant (TCR). This reduction is somewhat different than the DDH reduction, in that it is a reduction to a computation problem rather than to a decision problem. In terms of using the tool, there is only one game that is generated from the reduction template, and the goal is to show that $\mathbf{CCA}_5$ is equivalent to that game. (In the previous reduction there were two games, depending on the value of b_{DDH} and the goal was to prove that $\mathbf{CCA}_1$ is equivalent to one of these games and $\mathbf{CCA}_2$ is equivalent to the other.) Also, the "thing to be justified" by the reduction is the setting of some bad-event flag, rather than the difference between two games.

The reduction to TCR is handled also via a template, which would automatically generate a TCR code template, as described in Figure 13. Again the user can transform the game $\mathbf{CCA}_5$ via permissible transformations in order to fill the TCR template. In this case the transformation is pretty trivial, as game $\mathbf{CCA}_5$ can almost immediately be cast as an instance of the TCR template. Specifically, the first call to the collision-finder is substituted for the code in lines 002–005 (that generates $a, \hat{a}, c$), then the hashing key hk if chosen in line 006, and the rest of the code is substituted for the second call to the collision-finder, looking for s such that $v[s] = v$ but $(a[s], \hat{a}[s], c[s]) \neq (a, \hat{a}, c)$. Once the reduction is complete, the user can eliminate the designation of bad_2 as a bad-event flag (but keep it in the code, since it is used it in the next step).

Completing the proof. The rest of the proof is done by showing that the probability of the flag bad_1 being set but not bad_2 is small. This may be done on paper, but in this case it should be even possible to push the argument via the tool itself. First the user adds to the code a variable

$$r' \leftarrow u(x + vy) + (u - u')w(x_2 + vy_2) \bmod q$$

and then replaces the lines

008a $x_2, y_2 \xleftarrow{\$} [0, q-1]$, $x_1 \leftarrow x - wx_2 \bmod q$, $y_1 \leftarrow y - wy_2 \bmod q$
008' $r' \leftarrow u(x + vy) + (u - u')w(x_2 + vy_2) \bmod q$
009 $d \leftarrow a^{x_1 + vy_1} \cdot \hat{a}^{x_2 + vy_2}$

by the following lines (which are equivalent as long as $u \neq u'$ and $w \neq 0$):

008 $r' \xleftarrow{\$} [0, q-1]$, $d \leftarrow g^{r'}$
009a $y_2 \xleftarrow{\$} [0, q-1]$
009b $x_2 \leftarrow \frac{r' - u(x+vy)}{w(u'-u)} - vy_2 \bmod q$
009c $x_1 \leftarrow x - wx_2 \bmod q$, $y_1 \leftarrow y - wy_2 \bmod q$

Now, since the variable d no longer depends on the x, y's, lines 009a–c can be moved to the area of the code where the flag bad_1 is computed (and also push down the code that computes bad_2).

The user adds to the code the variables $u[s] \leftarrow DL_g(a[s])$, $u'[s] \leftarrow DL_{\hat{g}}(\hat{a}[s])$, and $r'[s] \leftarrow DL_g(d[s])$, replaces the test $a[s]^w \stackrel{?}{=} \hat{a}[s]^w$ in line 043 with $u'[s] \stackrel{?}{=} u[s]$, and also replaces the test $d[s] \stackrel{?}{=} a[s]^{x_1+v[s]y_1} \cdot \hat{a}[s]^{x_2+v[s]y_2}$ by the equivalent test $r'[s] \stackrel{?}{=} u[s](x+v[s]y) + (u[s]-u'[s])w(x_2+v[s]y_2)$. Call the resulting game $\mathbf{CCA}_6$. The relevant lines of code that describe the computation of the flags bad_2 and bad_1 are as follows:

040 $\quad y_2 \stackrel{\$}{\leftarrow} [0, q-1],\ \ x_2 \leftarrow \frac{r'-u(x+vy)}{w(u'-u)} - vy_2 \bmod q$
041 $\quad$ **for** $s \leftarrow 1$ **to** $2\mathsf{t}_{\mathsf{CCA}}$ **do**
042 $\qquad u[s] \leftarrow DL_g(a[s]),\ \ u'[s] \leftarrow DL_{\hat{g}}(\hat{a}[s]),\ \ r'[s] \leftarrow DL_g(d[s])$
043 $\qquad$ **if** $v[s] = v$ **then** $\mathsf{bad}_2 \leftarrow \mathsf{true}$
044 $\qquad$ **else if** $u[s] \neq u'[s]$ **and** $r'[s] = u[s](x+v[s]y) + (u[s]-u'[s])w(x_2+v[s]y_2)$ **then** $\mathsf{bad}_1 \leftarrow \mathsf{true}$

Now the tool can substitute the value of x_2 in the test on line 044 and also verify that under the conditions $v \neq v[s]$ and $h[s] \neq u'[s]$, the coefficient of y_2 is non-zero. Again, the tool can be endowed with a rule that lets it assign a probability of $1/q$ for this test, and since it appears in a loop that is executed at $2\mathsf{t}_{CCA}$ times, it has total probability of $2\mathsf{t}_{CCA}/q$.

4.2 Re-cap of the Cramer-Shoup security proof

Above I demonstrated in excruciating detail how one can use an automatic tool to re-generate the proof of security in [CS01]. On a high level, this involves manipulating games using both code movement techniques as well as algebraic manipulations. It should be noted that there exist automated tools that do both kinds of manipulations. Clearly, every optimizing compiler must perform code movement. Also, for example the software tool Mathematica [Wol03] does many symbolic algebraic manipulations, some of which are much more complex than the simple operations that I described above.

The overall structure of the proof is illustrated graphically in Figure 14. Very roughly, it starts from the CCA-security game (instantiated with the CS scheme) and proceed via steps, at each step modifying the game somewhat. Some of these steps can be shown to preserve the distribution over the adversary's view (or alter it only by a small amount), while other steps are justified by reductions to hard problems (i.e., DDH or the target-collision-resistance of the hash function in use). The original game depends on a binary parameter b_{CCA} (that determines which of the messages is encrypted to form the target ciphertext). At some point in the sequence (CCA4), the analysis arrives at a game that no longer depends on the parameter b_{CCA}, and the output can be eliminated. However, at that point there are some "bad events", such that the new game maintain the probability distribution of the previous games only so long as these events do not happen. The rest of the proof then is devoted to bound the probability of these bad events, either by making computational assumptions (cf. the transformation from CCA4 to CCA5) or via information-theoretic arguments. Once all the bad events are taken care of, the resulting game has no output and no bad flags, hence all the code in that game becomes irrelevant and can be eliminated.

It is important to stress that all the information in Figure 14 can be "known" to the tool, as explained above. Hence, in principle it is possible to have the tool output this picture.

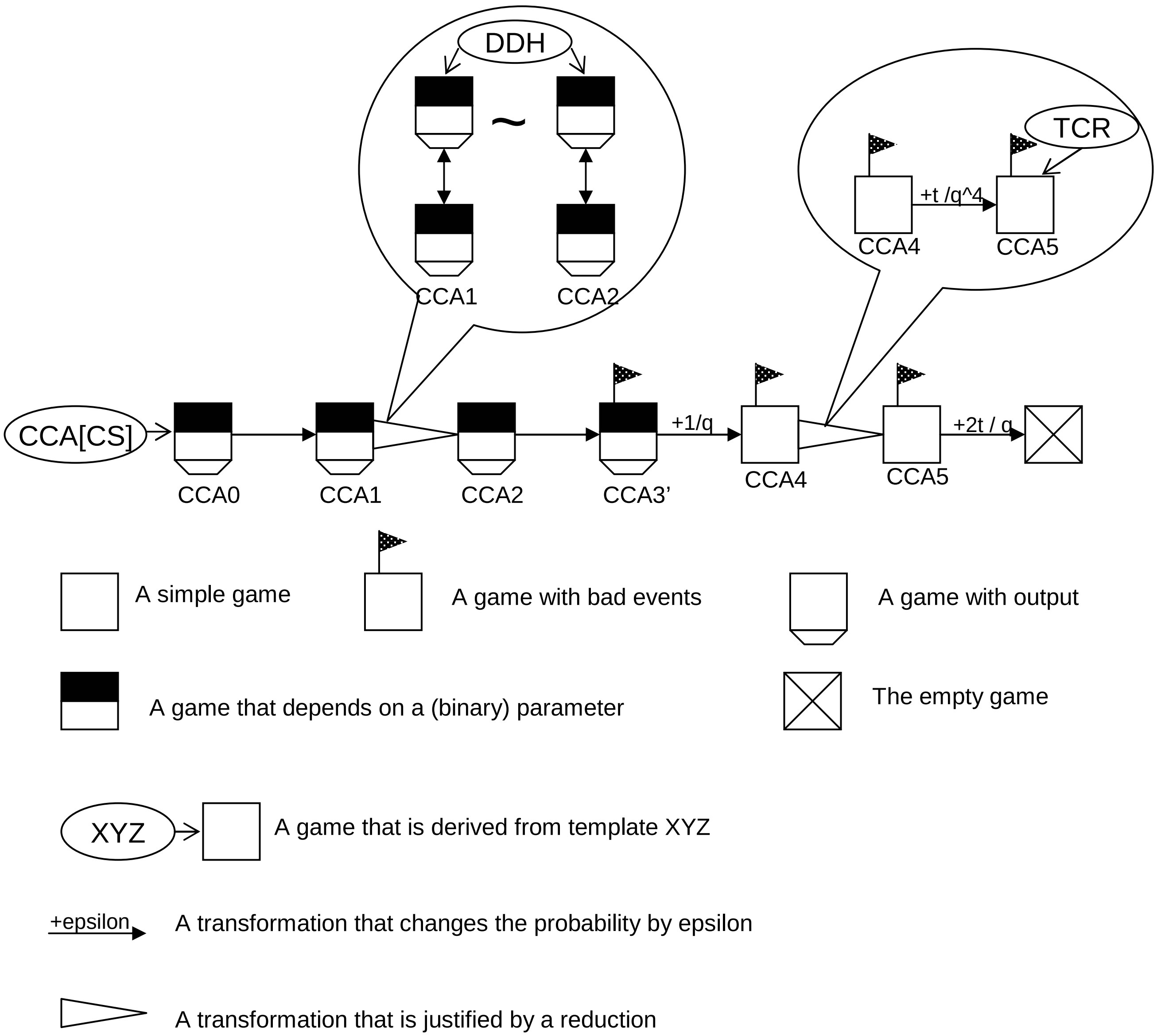

Figure 14: A pictorial description of the Cramer-Shoup proof of security

5 Building the tool

In this section I sketch some ideas about how this tool can be built. The tool will consists of four major components: basic engine, a library of transformation and rules, a library of templates, and user interface. Below I briefly discuss each of these components.

Basic engine. The basic engine is what manipulates the code. In essence, this component should be similar to an optimizing compiler. It is able to read the code, parse it into statements, figure out the control-flow graph and the dependencies between variables, etc. Most importantly, it should allow the user to manipulate the code, as long as the relevant dependencies are not violated. This includes code movement (e.g., switching the order of two statements that do not depend on each other), variable substitution (e.g., replacing $x \leftarrow a+b, y \leftarrow x+c$ by $x \leftarrow a+b, y \leftarrow a+b+c$), and perhaps some other simple transformations. The user should also be allowed to add or eliminate "dead code". Namely, at any point the tool will have some set of variables that are considered "target variables"[4] and the user will be allowed to add and remove statements, as long as these statements do not effect these target variables.

Library of transformations. In addition to the very low-level transformations that are "hard wired" in the engine, the tool will also have other transformations that represent common arguments that we make in our proofs (such as algebraic manipulations and claims that "doing X induces the same probability distribution as doing Y"). These transformations may depend on the data-types of the variables that are involved, and they may also carry some penalty. Some of the transformations in the library include the following:

- *Adding bad-event flags.* The user will be allow to add boolean variables that are designated as bad-event flags, and must be initialized to `false` and only have assignment to `true` in the code. The effect of adding them is that they are immediately designated as "target variables" as above, and also any statement that is only executed in branches where these variables are set to `true` is immediately designated as "dead code". (Note that there may be some types of statements that introduce their own implicit bad-event flags. For example, the statement $y \leftarrow a/x \bmod q$ implicitly introduces a bad-event flag that is set when $x = 0$.)

- *Coin fixing.* In most (perhaps all) of the interesting games, there will be a stub function that is designated as an *adversary subroutine.*[5] This is a completely unspecified subroutine (akin to an `extern` function in "C") that is called in the game. The coin-fixing transformation is a special rule to convert variables from being used as input to that routine to become output of it. For this to be a permissible transformation, the game *must not* have the output as a "target variable". (Namely, it must be the case that the only interesting aspects of this game is the probability that some bad-event flags are set.)

  Then, the statements $x \leftarrow f(y_1, \cdots, y_n)$, $z \leftarrow \mathbf{Adversary}(x, y_1, \cdots, y_n, \cdots)$ can be replaced by $(x, z) \leftarrow \mathbf{Adversary}(y_1, \cdots, y_n, \cdots)$. A special case of this (which is probably the most

[4] These will be the bad-event flags and the output of the game.

[5] Perhaps it is possible to have more than one adversary subroutine, but I cannot think of any example where this is needed.

useful) is that the statements $x \xleftarrow{\$} \{0,1\}^n$, $z \leftarrow \textbf{Adversary}(x, \cdots)$ can be replaced by $(x, z) \leftarrow \textbf{Adversary}(\cdots)$.[6]

- *Simple algebraic manipulations.* With every data type that is supported by the tool, there will be a set of rules that specify what transformations are allowed for statements that involve this data type. For example, many data types represent mathematical objects with operations that are commutative, associative, and distributive. So the tool must know that the statement $u \leftarrow v(x + yz)$ is equivalent to $u \leftarrow xv + yvz$.

- *Distribution-preserving transformations.* The tool will also have rules that allow the user to change the way randomness is introduced without changing the resulting probability space. For example, there is a rule saying that when x, y, c are of type n-bit string (with n an integer parameter) the statements $x \xleftarrow{\$} \{0,1\}^n,\ y \leftarrow x \oplus c$ are equivalent to $y \xleftarrow{\$} \{0,1\}^n,\ x \leftarrow y \oplus c$.

  As another example, if $\mathbf{G}$ is of type prime-order group (with the integer parameter q as its order), and if g, h are of type elements of $\mathbf{G}$, and if w is of type residue mod q, then the statements $g \xleftarrow{\$} \mathsf{NotOne}(\mathbf{G}),\ h \xleftarrow{\$} \mathbf{G},\ w \leftarrow DL_g(h)$ are equivalent to $g \xleftarrow{\$} \mathsf{NotOne}(\mathbf{G})$, $w \xleftarrow{\$} [0, q-1]$, $h \leftarrow g^w$.

  (Here $\mathsf{NotOne}(\mathbf{G})$ is supposed to represent removing the unit of $\mathbf{G}$. From the tool's perspective, however, NotOne is simply some unitary operation on prime-order groups that has some rules associated with it. One of these rules is that the transformation from above is permissible. Some other rules will be described in some of the examples below.)

- *Nearly-distribution-preserving transformations.* There are also transformations that do not quite maintain the distribution of variables, but only induce a small deviation. One example is changing between $g \xleftarrow{\$} \mathsf{NotOne}(\mathbf{G})$ and $g \xleftarrow{\$} \mathbf{G}$, that only induce a deviation of $1/q$, where q is the order of $\mathbf{G}$. A more interesting example is replacing the two statements

  001 $\quad x \xleftarrow{\$} \{0,1\}^n$
  002 $\quad$ do-something-with-x

  with

  001 $\quad x \xleftarrow{\$} \{0,1\}^n$
  002 $\quad$ **if** $x \neq 0^n$ **then** do-something-with-x
  002 $\quad$ **else** do-something-else

  that induces a deviation of 2^{-n}. This rule can be derived in the tool as follows: For each variable, the tool will keep not only a data type, but also some other tags, one of which is "the distribution of that variable" in different points in the code. This tag can be simply an integer (corresponding to the (min)entropy of that variable), or it can be a more complex structure that holds more information about the distribution of that variable.[7] Either way, the tool will have a rule saying that if two variables x, y are independent (in the sense of control-flow of the code) then the condition $x = y$ has probability at most 2^{-m}, where m is the larger between the min-entropy of x, y. Using such a rule, the tool can deduce that the condition **if** $x \neq 0^n$ in line 002 is satisfied with all but probability of 2^{-n}.

[6] One can discern more general conditions that allow coin-fixing, but I see no reason for the extra generalization unless there are some interesting proofs that actually utilize them.

[7] The decision of how much information needs to be kept on the distribution of variables depends on the extent to which common proofs utilize this information. In the two case studies in this paper, it was always sufficient to keep only the min-entropy of the variables. I believe that this is the case for the vast majority of proofs in the literature, but probably not for all of them.

Perhaps the most useful nearly-distribution-preserving transformations are bad-event elimination, as discussed in Section 2.1. Also, many nearly-distribution-preserving transformations can actually be viewed as a special case of bad-event elimination. For example, the transformation from $g \xleftarrow{\$} \mathbf{G}$ to $g \xleftarrow{\$} \mathsf{NotOne}(\mathbf{G})$ can be viewed as first setting $g \xleftarrow{\$} \mathbf{G}$, **if** $g = 1$ **then** $\mathsf{bad} \leftarrow \mathsf{true}$ and then eliminating the flag bad. (On the other hand, I don't see any way to view the transformation in the other direction as an instance of bad-flag elimination).

It is likely that many of the rules from above can be recognized in the tool as special cases of some more general rules. Indeed, creating this library of rules and finding the "right level of abstraction" is one of the hardest challenges in building this tool. On one hand we want to have rules that are high-level enough to be useful in many different proofs, and on the other hand the rules should be low-level enough so that it is easy to check automatically that a given transformation is indeed justified by one of the rules.

Since this library represents the "types of arguments" that we use in out proofs, it is likely that it will have to grow with time, as we discover new arguments. Whenever a new proof comes along that uses an argument that is not yet in the library (and if this argument is "general purpose" enough to be useful in other proofs), one could just add a rule to the library that describes this new argument, and from then on be able to incorporate the new argument in proofs that the tool can check.

Templates. Just like the library of rules represents the type of arguments that we use in our proofs, the library of templates represents the computational assumption that we use (and the definitions that we want to realize). For example, there will be two templates for CPA-secure encryption. One is a template for "construction of CPA-secure scheme", and it will be utilized when a new encryption scheme is described and one wants to prove that it is CPA secure. The other template is for "reduction to CPA security", and it is utilized when an encryption scheme is used as a building block in a larger protocol, and the security of the larger protocol relies on the CPA-security of the encryption.

There may also be higher-level templates that represent "frameworks". For example, a UC template will let the user specify a UC functionality, and automatically generate the "ideal-world game" of the UC framework with the environment and let the user fill in the details of the simulator. Similarly, the same template will also let the user specify the protocol and automatically generate the "real-world game" with the same environment.

Hopefully, writing these templates can be done as a distributed community development effort. For example, whenever people comes up with a new computational assumption (or a new cryptographic definition), they will also write a template to introduce this assumption/definition to the tool, thereby allowing other users of the tool to use the same assumption/definition.

User interface. This will be no doubt the most complicated (and most crucial) component of the tool. Just like any other software product, the usefulness of this tool will depend crucially on the willingness of the customers (in this case the cryptography community) to use it. Not knowing much about UI design, I cannot really guess how this would be done. Below I just list a few considerations that will have to be addressed in the design.

- It should be easy for the user to input into the tool a description of the scheme to be analyzed.

This means that (a) it should be easy to write code for the tool, and (b) the tool must recognize high-level constructs that cryptographers want to use. For example, it must be easy enough to input into a tool a step that says "prove $x \in L$ in zero-knowledge" (where x, L are specified somewhere, and a "zero-knowledge proof" is included in the template library).

- It should be easy enough for the user to view the code and transform it. For example, maybe moving code can be done with mouse-drag. Also, it should be easy for the user to tell the tool things like "now I want to do coin-fixing" or "this transformation should be justified by reduction to DDH".
- The tool should be able to output the code of a game in LaTeX format, so that the user will be able to prove on paper arguments that are not be represented in the tool. The tool should also be able to output, say, a "C" code of the scheme to be analyzed (in case this is a practical scheme that people actually want to use, and it needs test-vectors and such like).
- The tool should be able to produce some pictorial representation of the proof (such as described in Figures 5 and 14), to help a human verifier check the proof.

5.1 What would it take to create this tool?

It is quite clear (to me, at least), that creating a tool as sketched in this report is technically possible. It is just as clear, however, that this is a major project. My guess is that the effort involved is similar to creating a new programming language, complete with development environment and run-time support. It also requires quite a bit of cooperation between experts in different areas (at least compilers, user-interface and cryptography).

It seems that although technically possible, this tool does not have a very appealing "business case". Why would anyone wants to invest so much effort in a tool that only serves such a small community? (The entire community of people who write crypto proofs is unlikely to be more than 200 people.) I would still argue that it is worth investing in it, for the following reasons:

- *Good tools find other uses.* Although I only considered the use for crypto proofs, I expect that many of the techniques that are described here will find uses also in other places.
- *Interesting research.* I also believe that some of the aspects that are involved in creating this tool will lead to interesting results. For example, I believe that designing the library of rules for this tool could shed light on good representations of human knowledge in computer systems.
- *The need is real.* My feeling is that we are quickly approaching the threshold where most of our crypto proofs are no longer verifiable. Something has got to give, and using computer-aided tools is an approach that so far did not receive the attention that it deserves.

Also, I believe that if a tool like that is built well, it will be adopted and used by many. Wouldn't you like to be cited by half of the papers appearing in CRYPTO 2010? Here is your chance...

6 Conclusions

In this paper I argued for creating a complier-like tool that can help us prove the security of cryptographic scheme. I demonstrated how such as tool can be used to reconstruct two crypto proofs from the literature, and mused about how such a tool can be built.

Before concluding, I would like to point out that there are several common arguments in crypto proofs that are not present in the two proofs that I chose for my case studies, and therefore are not discussed in this report. In particular, I didn't address at all proofs in the random oracle model, or other non-standard models. However, it seems that working in other models is unlikely to be different than working in the standard model, as long as you can describe all the components in the model via code. In particular, as far as I can tell working in the random-oracle model does not induce any new issues that I did not already address in the standard model. (You just have another efficient procedure in the game to implement the random oracle.)

Also, I did not address in this report common 1-out-of-n hybrid arguments (i.e., reductions that are not tight). These will have to be represented by other transformations in the library. The transformations will probably be similar to the nearly-distribution-preserving variety, except that they change things by a multiplicative factor rather than an additive one.

Finally, all the arguments that I considered here are "black-box arguments" (in the sense that subroutines are only used by calling them, never by doing anything else with their code). I did not think about non-black-box arguments, but I speculate that dealing with them may necessitate some "reflection" mechanism, similar to what exists in interpreted programming languages.

References

[BR04] Mihir Bellare and Phil Rogaway. The game-playing technique. Cryptology ePrint on-line arcive, `http://eprint.iacr.org/2004/331`, 2004.

[CS01] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2001. Preliminary version in Crypto'98.

[Hal04] Shai Halevi. EME*: extending EME to handle arbitrary-length messages with associated data. In *5th International Conference on Cryptology in India, INDOCRYPT'04*, volume 3348 of *LNCS*, pages 315–327. Springer, 2004. Full version available on the ePrint archive, `http://eprint.iacr.org/2004/125/`.

[HR03] S. Halevi and P. Rogaway. A tweakable enciphering mode. In D. Boneh, editor, *Advances in Cryptology – CRYPTO '03*, volume 2729 of *LNCS*, pages 482–499. Springer, 2003. Full version available on the ePrint archive, http://eprint.iacr.org/2003/148/.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 33–43, 1989.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint on-line arcive, `http://eprint.iacr.org/2004/332`, 2004.

[Wol03] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc., 5 edition, August 2003. See also `http://www.wolfram.com/products/mathematica/`.