Automated Proofs of Pairing-Based Cryptography

Gilles Barthe
IMDEA Software Institute
Madrid, Spain
gilles.barthe@imdea.org

Benjamin Grégoire
INRIA
Sophia-Antipolis, France
benjamin.gregoire@sophia.inria.fr

Benedikt Schmidt
IMDEA Software Institute
Madrid, Spain
benedikt.schmidt@imdea.org

ABSTRACT

Analyzing cryptographic constructions in the computational model, or simply verifying the correctness of security proofs, are complex and error-prone tasks. Although computer tools have significant potential to increase confidence in security proofs and to reduce the time for building these proofs, existing tools are either limited in scope, or can only be used by formal methods experts, and have a significant overhead. In effect, it has remained a challenge to design usable and intuitive tools for building and verifying cryptographic proofs, especially for more advanced fields such as pairing-based or lattice-based cryptography.

This paper introduces a formal logic which captures some key reasoning principles in provable security, and more importantly, operates at a level of abstraction that closely matches cryptographic practice. Automatization of the logic is supported by an effective proof search procedure, which in turn embeds (extended and customized) techniques from automated reasoning, symbolic cryptography and program verification. Although the logic is general, some of the techniques for automating proofs are specific to fixed algebraic settings. Therefore, in order to illustrate the strengths of our logic, we implement a new tool, called AutoG&P, which supports extremely compact, and often fully automated, proofs of cryptographic constructions based on (bilinear or multilinear) Diffie-Hellman assumptions. For instance, we provide a 100-line proof of Waters' Dual System Encryption (CRYPTO'09), and fully automatic proofs of Boneh-Boyen Identity-Based Encryption (CRYPTO'04). Finally, we provide an automated tool that generates independently verifiable EasyCrypt proofs from AutoG&P proofs.

Categories and Subject Descriptors

F.3 [**Reasoning about Programs**]: Logics of programs

Keywords

automated proofs; provable security; public-key encryption

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813697.

1. INTRODUCTION

Formal verification tools have the ability to deliver high-integrity artefacts; they can also increase productivity of artefact designers, provided they can achieve reasonable trade-offs between benefits and costs of formal verification. Although there is a lack of data and metrics to measure gains in productivity, general-purpose software is one area where appropriate trade-offs have been identified and where formal verification has achieved significant benefits. In contrast, it has proved more challenging to leverage the potential benefits of formal verification in security-related areas, partly because formalizing properties of interest and adequate models to reason about them is hard. One notable exception where formal verification has been used very successfully is the security of cryptographic protocols in the Dolev-Yao or symbolic model, for which numerous automated tools have been developed [12, 21, 35]. However, proofs in the symbolic model are restricted to cryptographic protocols (rather than primitives) and deliver weaker guarantees than proofs in the computational model. The question then arises whether one can design verification tools for analyzing the security of cryptographic protocols and primitives in the computational model. Significant progress has been made over the last ten years, and tools like CertiCrypt [9], CryptoVerif [13], EasyCrypt [8], and more recently FCF [34], have been used to verify emblematic case studies. However, automated proofs of cryptographic primitives remain out of scope, and it is also a challenge to support proofs which follow the same structure and level of abstraction as pen-and-paper proofs from the literature. As a consequence, the scopus of formally verified proofs (almost) completely elides some of the most important developments in the field, such as pairing-based or lattice-based cryptography (the only exception is a formal proof of chosen-plaintext security of Boneh-Franklin Identity Based Encryption in the random oracle model [10]) and adoption of formal proofs by the cryptographic community has been limited.

Contributions. The main contribution of this paper is a formal logic to reason about the concrete security of cryptographic constructions directly in the computational model. The distinguishing characteristics of our logic is its ability to adhere to a level of abstraction that is close to the one used in pen-and-paper proofs from the cryptography literature and to deliver compact and intuitive formal proofs, using a core set of rules for bridging steps, failure events, reductions, or hybrid arguments. The logic is similar in spirit to CIL [6], but is instantiated to a functional programming

Type	types of expressions	
t ::=	$\mathbb{B}$	boolean value
|	$\mathbb{BS}_l$	bitstring of length $l \in \mathsf{Len}$
|	$\mathbb{G}_i$	cyclic group with $i \in \mathsf{GName}$
|	$\mathbb{F}_q$	prime field of order q
|	$t \times \ldots \times t$	tuple

Figure 1: Grammar for types.

language. Moreover, the logic is supported by an effective proof search procedure which applies high-level rules built on top of the core rules and exploits (often extended and customized) techniques from automated reasoning (e.g. algorithms for equational reasoning), symbolic cryptography (e.g. algorithms for deducibility), and program verification (e.g. strongest post-condition calculus) for automatically discharging proof obligations. The proof search procedure is proof-producing, in the sense that the proofs it constructs are elaborated into elementary proofs in which all inferences are performed using the core rules of the logic. Such elementary proofs are then translated into EasyCrypt, to obtain foundational proofs in which all inference steps are explained in terms of relational program logics.

Although the logic is general, some of the techniques used for automating proofs, in particular those related to equational reasoning, are specific to an algebraic setting. Because of its importance in modern cryptography, we focus on pairing-based cryptography, and implement a new tool, called AutoG&P, able to analyze the security of pairing-based constructions. Using AutoG&P, we provide the first formal proofs of pairing-based constructions in the standard model, including a fully automated proof of Boneh-Boyen Identity Based Encryption [14] and a short (<100 lines) proof of Waters' Dual System Encryption [36]. Our proofs show the feasibility of formal proofs for pairing-based cryptography. Moreover, and quite interestingly, our proofs also suggest that AutoG&P provides an adequate level of abstraction for transforming automatically proofs of security from Type I to Type III settings, as investigated in [4, 3], as the same high-level proof can be used to prove the security of the Boneh-Boyen IBE scheme in the two settings.

2. NOTATION

In this paper, we consider expressions, programs, and security experiments as terms and make use of the following operations on terms. We use $t|_p$ to refer to the subterm of the term t at position p and $t\{t'\}_p$ to refer to the result of replacing the term at position p with t'. If p is a position in a sequence, we abuse notation and write $t\{\}_p$ to remove the element in the sequence at position p. We write $t\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ to denote the result of substituting x_i by t_i in t. A context C is a term with a distinguished variable $\Box$ which denotes a hole that can be filled in by an arbitrary term. We use $C\{t\}$ to denote the term obtained by plugging t into $C's$ hole.

3. LANGUAGE

In this section, we define the syntax and semantics of expressions, games, security experiments, and judgments.

3.1 Types and expressions

The set Type of *types* is defined by the grammar given in Figure 1 where Len denotes a finite set of length variables and GName denotes a finite set of group names. We assume given disjoint infinite sets Var_t of *typed variables* and define Var as the union of these sets. We also assume given sets Fsym of function symbols and $\mathsf{Emap} \subseteq \mathsf{GName} \times \mathsf{GName} \times \mathsf{GName}$ and $\mathsf{Isom} \subseteq \mathsf{GName} \times \mathsf{GName}$ defining the types of modeled bilinear maps and isomorphisms between the groups. Given these sets, we define the corresponding signature Sig in Figure 2. The set Expr of *expressions* consists of all terms built over Sig. We say an expression is *efficient* if it does not contain log.

A *group setting* $\mathcal{G} = (q, \{G_i\}_{i \in \mathsf{GName}}, \{\hat{e}_j\}_{j \in \mathsf{Emap}}, \{\phi_j\}_{j \in \mathsf{Isom}})$ consists of a prime q, an indexed set of cyclic groups G_i of order q, an indexed set of bilinear maps $\hat{e}_{r,s,t} : G_r \times G_s \to G_t$, and an indexed set of isomorphisms $\phi_{r,s} : G_r \to G_s$. We assume that the structure defined by Emap and Isom does not contain cycles that would allow for an unbounded number of multiplications in the exponent. Our formalism can be used to model bilinear groups of Type I, Type II, Type III, and (leveled) k-linear groups [15, 17, 28]. An *interpretation* $\mathcal{I}$ consists of a group setting $\mathcal{G}$, a mapping from length variables to natural numbers, and a mapping from function symbols to functions. We write $\mathcal{I}(l)$ to denote the length assigned to l and $\mathcal{I}(h)$ to denote the function assigned to h.

3.2 Equivalence of expressions

We define an equivalence relation on expressions based on satisfaction in first-order logic. The expressions e and e' are equivalent modulo $\mathcal{E}$, written $e =_{\mathcal{E}} e'$, if $\mathcal{E} \models e = e'$. Here, $\mathcal{E}$ denotes the axioms for our signature consisting of the field axioms for $\mathbb{F}_q$, the (bilinear) group axioms for $\mathbb{G}_i$, and the usual axioms for congruence, the logical operators and the bitstring operators. We consider inversion in $\mathbb{F}_q$ as underspecified, i.e., 0^{-1} is some arbitrary fixed value in $\mathbb{F}_q$ and we can only simplify $x * x^{-1}$ to 1 if $x \neq 0$ holds. We use $\Gamma \models e =_{\mathcal{E}} e'$ to denote $(\Gamma, \mathcal{E}) \models e = e'$, i.e., $\mathcal{E}$ is extended with additional axioms Γ. We assume the set of axioms Γ consists of (in)equalities on expressions.

We use contexts to express algorithms that can be defined using the signature Sig. We say a context C is *ground* if C does not contain any variables except $\Box$. We write $e \vdash^C_{\mathcal{E}} e'$ if $C\{e\} =_{\mathcal{E}} e'$ for a ground context C. We write $e \vdash_{\mathcal{E}} e'$ if there exists a context C such that $e \vdash^C_{\mathcal{E}} e'$. Similarly, we write $\Gamma \models e \vdash^C_{\mathcal{E}} e'$ if $\Gamma \models C\{e\} =_{\mathcal{E}} e'$ and $\Gamma \models e \vdash_{\mathcal{E}} e'$ if there exists such a context C. For example, it does not hold that $a * b \vdash_{\mathcal{E}} b$, but $a \neq 0 \models a * b \vdash_{\mathcal{E}} b$ holds as witnessed by the context $C = \Box / a$.

3.3 Games

A *game* is a sequence of game commands. A *game command* is a let binding, a random sampling, an assertion, or an adversary call. For each adversary call, the provided oracles are defined inline. An *oracle definition* consists of the oracle symbol, the parameters, a sequence of oracle commands, and the return value. An *oracle command* is a let binding, a random sampling, or a guard that ensures that $\perp$ is returned unless the given test succeeds. The grammars O for oracle definitions and gc for game commands are given in Figure 3. In the grammars, we use typed oracle symbols $\mathsf{o} \in \mathsf{Osym}$ and typed adversary symbols $\mathsf{A} \in \mathsf{Asym}$. The scope of variables bound in the body of games extends to succeeding or-

Sig	Signature for expressions	
$x : t$		variables for $x \in \mathsf{Var}_t$
$h(\_) : t_1 \to t_2$		function symbols $h \in \mathsf{Fsym}$
$(\_, \ldots, \_) : t_1 \times \ldots \times t_k \to (t_1 \times \ldots \times t_k), \quad \pi_i : t_1 \times \ldots \times t_k \to t_i$		tuple construction and projection
$g_i : \mathbb{G}_i$		generator of $\mathbb{G}_i$
$\_ * \_ : \mathbb{G}_i \times \mathbb{G}_i \to \mathbb{G}_i, \quad \_/\_ : \mathbb{G}_i \times \mathbb{G}_i \to \mathbb{G}_i$		multiplication and division in $\mathbb{G}_i$
$(\_)^{(\_)} : \mathbb{G}_i \times \mathbb{F}_q \to \mathbb{G}_i, \quad \log : \mathbb{G}_i \to \mathbb{F}_q$		exponentiation and discrete log in $\mathbb{G}_i$
$\hat{\mathsf{e}}_i : \mathbb{G}_{i_1} \times \mathbb{G}_{i_2} \to \mathbb{G}_{i_3}, \quad \phi_j : \mathbb{G}_{j_1} \to \mathbb{G}_{j_2}$		bilinear map ($i \in \mathsf{Emap}$), isomorphism ($j \in \mathsf{Isom}$)
$0 : \mathbb{F}_q, \quad \_ + \_ : \mathbb{F}_q \times \mathbb{F}_q \to \mathbb{F}_q, \quad -(\_) : \mathbb{F}_q \to \mathbb{F}_q$		additive group operations for $\mathbb{F}_q$
$1 : \mathbb{F}_q, \quad \_ * \_ : \mathbb{F}_q \times \mathbb{F}_q \to \mathbb{F}_q, \quad (\_)^{-1} : \mathbb{F}_q \to \mathbb{F}_q$		multiplicative group operations for $\mathbb{F}_q$
$0_l : \mathbb{BS}_l, \quad \_ \oplus \_ : \mathbb{BS}_l \times \mathbb{BS}_l \to \mathbb{BS}_l$		operations on bitstrings $\mathbb{BS}_l$
$\_ = \_ : t \times t \to \mathbb{B}, \quad (\_ ? \_ : \_) : \mathbb{B} \times t \times t \to t$		equality and if-then-else
$\neg \_ : \mathbb{B} \to \mathbb{B}, \quad \_ \wedge \_ : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$		negation and conjunction

Figure 2: Signature for expressions.

acle definitions. In assertions, we support *event expressions* $ev \in \mathsf{Expr}^{ev}$, which are also defined in Figure 3. The quantifications range over the parameters used in adversary queries. For example, the assertion $\mathsf{assert}(\forall c \in Q_{Dec_1}.\, c \neq c^*)$ expresses that c^* is a ciphertext that has not been queried to the decryption oracle Dec_1. We use $osym(G)$ to denote all oracle symbols occuring in a game G and $asym(G)$ to denote all adversary symbols occuring in a game G. We say a game G is *well-formed* if it is well-typed, does not contain free variable occurences, all oracle and adversary symbols are distinct, and all binders bind distinct variables. To simplify the presentation, we allow for at most one exceptional value in random samplings. This restriction can be lifted, but care must be taken to ensure that the types are large enough such that the support of the resulting distribution is not empty.

We say a tuple $S = (\mathcal{I}, \{\delta_{\mathsf{o}}\}_{\mathsf{o} \in osym(G)}, \{\mathcal{A}_{\mathsf{A}}\}_{\mathsf{A} \in asym(G)})$ is a *G-setting* if $\mathcal{I}$ is an interpretation, δ is an indexed set of query bounds for oracle symbols occuring in G, and $\mathcal{A}$ is an indexed set of adversaries for adversary symbols occuring in G. Given such a setting, we can execute G as follows:

1. Compute generators of the groups G_i by random sampling or by applying the bilinear maps and isomorphisms to generators that have already been computed.

2. For each oracle o occuring in G, we initialize a counter variable c_{o} with zero and a query set variable Q_{o} with the empty set.

3. For a let binding $\mathsf{let}\ x = e$, we evaluate the expression e using the operations defined by $\mathcal{I}$ and store the result in the variable x.

4. For a random sampling $x \xleftarrow{\$} t \setminus a$, we evaluate a denoting the result with b, uniformly sample from the set $\mathcal{I}(t) \setminus \{b\}$ (where $\mathcal{I}(t)$ is set of values of type t), and store the result in the variable x.

5. For an assertion $\mathsf{assert}(ev)$, we evaluate ev and abort if the result is false.

6. For an adversary call $y \leftarrow \mathsf{A}(e)\ \mathsf{with}\ \vec{O}$, we evaluate e and call the adversary $\mathcal{A}_{\mathsf{A}}$ with the result as input. The adversary is provided with access to the implementations of the oracles $\vec{O}$. We allow the adversary procedures $\mathcal{A}_{\mathsf{A}}$ to share state.

The oracles are implemented as follows:

1. If $c_{\mathsf{o}} \geq \delta_{\mathsf{o}}$, then $\bot$ is returned. Otherwise, the counter c_{o} is increased, the query parameters are stored in Q_{o}, and the oracle body is executed.

2. Guards are treated similar to assertions, but instead of aborting, the value $\bot$ is returned to the adversary and execution continues normally.

To perform reductions, we want to be able to state that a game $G(\mathsf{B})$ is equivalent to a game $G'(\mathsf{A})$, capturing a cryptographic assumption, when the adversary A is instantiated in a certain way. Usually, A is instantiated with a simulator that simulates G to the original adversary B and uses B's output to break the assumption. Additionally, we might want to express the oracle bounds for A in G' in terms of oracle bounds for B in G.

To achieve this, we extend our syntax and semantics to support the instantiation of adversaries and oracle bounds. We extend the syntax of games as follows. A game G is either an uninstantiated game G as defined before, a game $G[\mathsf{n}_{\mathsf{o}} := f]$ where the bound for calls to o is instantiated by the polynomial f over oracle bounds, a game $G[\mathsf{A} := A]$ where the adversary symbol A is instantiated with the adversary definition A. Here, A is defined in some language that extends our language of games, e.g., the probabilistic programming language PWHILE [8] used in EasyCrypt. We allow such adversary definitions A to include calls to unspecified adversary procedures identified by adversary symbols B. To prevent cyclic definitions, we assume that $\mathsf{o} \notin osym(f)$ and $\mathsf{A} \notin asym(A)$. We extend $asym$ to return only uninstantiated adversary symbols:

$$asym(G) = \begin{cases} asym(G) & \text{if } G = G[\mathsf{n}_{\mathsf{o}} := f] \\ asym(G, A) \setminus \{\mathsf{A}\} & \text{if } G = G[\mathsf{A} := A] \end{cases}$$

$$
\begin{array}{rcll}
O & ::= & \mathsf{o}(x) = \{\vec{oc};\ \mathsf{return}\ e\} & \text{oracle definition} \\
oc & ::= & c & \text{ordinary command} \\
 & | & \mathsf{guard}(b) & \text{guard for } b \in \mathsf{Expr}_{\mathbb{B}} \\
gc & ::= & c & \text{ordinary command} \\
 & | & \mathsf{assert}(ev) & \text{assertion for } ev \in \mathsf{Expr}^{ev} \\
 & | & y \leftarrow \mathsf{A}(x)\ \mathsf{with}\ \vec{O} & \text{adversary call with oracles} \\
c & ::= & \mathsf{let}\ x = e & \text{let binding} \\
 & | & x \xleftarrow{\$} t \setminus a & \text{sample unif. from } t \setminus \{a\} \\
ev & ::= & e & \text{expression} \\
 & | & \exists\, b_1, \ldots, b_k.\, e & \text{there exist queries} \\
 & | & \forall\, b_1, \ldots, b_k.\, e & \text{for all queries} \\
b & ::= & x \in Q_{\mathsf{o}} & x \text{ ranges over queries}
\end{array}
$$

Figure 3: Grammars for oracle definitions and games.

$$
G^{\mathsf{BB}} = \left| \begin{array}{ll}
1: & c, d, h \xleftarrow{\$} \mathbb{F}_q;\ \mathsf{let}\ P = (g^c, g^d, g^h); \\
2: & i^* \leftarrow \mathsf{A}_1(); \\
3: & b \xleftarrow{\$} \mathbb{B};\ e \xleftarrow{\$} \mathbb{F}_q;\ \mathsf{let}\ C = (g^e, (P_2^{i^*} * P_3)^e); \\
4: & \mathsf{let}\ K_0 = \hat{\mathsf{e}}(P_1, P_2)^e;\ K_1 \xleftarrow{\$} \mathbb{G}_t; \\
5: & b' \leftarrow \mathsf{A}_2(P, C, (b ? K_0 : K_1))\ \mathsf{with} \\
 & \quad \mathsf{PrivKey}(i) = \{ \\
5.1: & \quad \mathsf{guard}(i \neq i^*); \\
5.2: & \quad r \xleftarrow{\$} \mathbb{F}_q; \\
5.3: & \quad \mathsf{return}\ \ (g^{(c*d+r*(d*i+h))}, g^r) \\
 & \quad \};
\end{array} \right|
$$

$$
G_\beta^{\mathsf{DBDH}} = \left| \begin{array}{l}
a, b, c \xleftarrow{\$} \mathbb{F}_q; \overline{t \xleftarrow{\$} \mathbb{F}_q}^{[\beta=1]} \\
b \leftarrow \mathsf{B}(g^a, g^b, g^c, \overline{\hat{\mathsf{e}}(g,g)^{a\,b\,c}}^{[\beta=0]}\ \overline{\hat{\mathsf{e}}(g,g)^{t}}^{[\beta=1]})
\end{array} \right|
$$

Figure 4: Game for IND-sID-CPA security of the Boneh-Boyen-IBKEM and game for DBDH assumption. The overlined expressions only occur for $\beta = 0$ (resp. $\beta = 1$).

Similarly, we extend *osym* to return only uninstantiated oracle symbols:

$$
osym(G) = \begin{cases} osym(G, f) \setminus \{\mathsf{o}\} & \text{if } G = G[\mathsf{n}_{\mathsf{o}} := f] \\ osym(G) & \text{if } G = G[\mathsf{A} := A] \end{cases}
$$

The definition of G-setting remains unchanged, but uses the extended versions of *asym* and *osym*. The instantiated oracle bounds and the instantiated adversaries are then computed from the bound polynomials and adversary definitions by using δ and A for the occuring oracle and adversary symbols.

3.4 Security experiments and judgments

A *security experiment* SE is a pair $[\ G : ev\ ]$ of a game G and an event expression ev. We say a security experiment is *well-formed* if G; $\mathsf{assert}(ev)$ is well-formed. We use $\Pr_S[\ G : ev\ ]$ to denote the probability that the execution of G in the setting S terminates without aborting and ev evaluates to true in the final memory.

We use the following grammar to define the set PExpr of *probability expressions*:

$$
\begin{array}{rcll}
P, P' & ::= & 0 & \text{zero} \\
 & | & P + P' & \text{addition} \\
 & | & \mathsf{n}_{\mathsf{o}} \times P & \text{security loss} \\
 & | & 1/|t| & \text{collision bound} \\
 & | & [\ G : ev\ ]_{\mathsf{succ}} & \text{success prob.} \\
 & | & [\ G : ev\ ]_{\mathsf{adv}} & \text{advantage prob.} \\
 & | & [\ G : ev\ ]_{[\ G' : ev'\ ]} & \text{distinguishing prob.}
\end{array}
$$

We call the subscripts succ, adv, and $[\ G' : c'\ ]$ *probability tags*. We say $S = (\mathcal{I}, \delta, \mathcal{A})$ is a P-setting if $\mathsf{dom}(\delta) = osym(P)$ and $\mathsf{dom}(\mathcal{A}) = asym(P)$. Given such a P-setting S, we define the probability function $prob_S(P)$ that assigns probabilities to probability expressions as follows:

$$
prob_S(P) = \begin{cases}
0 & \text{if } P = 0 \\
prob_S(P_1) + prob_S(P_2) & \text{if } P = P_1 + P_2 \\
\delta_{\mathsf{o}}\, prob_S(P_1) & \text{if } P = \mathsf{n}_{\mathsf{o}} \times P_1 \\
1/|\mathcal{I}(t)| & \text{if } P = 1/|t| \\
\Pr_S[\ SE\ ] & \text{if } P = [\ SE\ ]_{\mathsf{succ}} \\
\Pr_S[\ SE\ ] - \frac{1}{2} & \text{if } P = [\ SE\ ]_{\mathsf{adv}} \\
|\Pr_S[\ SE\ ] - \Pr_S[\ SE'\ ]| & \text{if } P = [\ SE\ ]_{SE'}
\end{cases}
$$

A *probability judgment* J is a pair $P \preceq P'$ of probability expressions. A judgment $P \preceq P'$ is *valid* if

1. $asym(P') \subseteq asym(P)$,
2. $osym(P') \subseteq osym(P)$, and
3. for all P-settings S, it holds that $prob_S(P) \leq prob_S(P')$.

Conditions (1) and (2) ensure that all adversary and oracle symbols that occur only in P' must be instantiated.

Example 1. For the games given in Figure 4, we can express the probability associated to the IND-sID-CPA security of the Boneh-Boyen-IBKEM as $[\ G^{\mathsf{BB}} : b = b'\ ]_{\mathsf{adv}}$ and we can express the DBDH assumption as

$$
[\ G_0^{\mathsf{DBDH}} : b\ ]_{[\ G_1^{\mathsf{DBDH}} : b\ ]}.
$$

We consider the key encapsulation mechanism (KEM) variant of the Boneh-Boyen identity-based encryption scheme [14] to simplify the presentation of our example. The game G^{BB} proceeds as follows. In line 1, the master secret key and the public parameters are computed. In line 2, the adversary must choose the challenge identity. In lines 3 and 4 the challenge encapsulation C, the corresponding session key K_0, and a random session key K_1 are computed. In line 5 the adversary is called and must guess if he received the real or a random session key. He is provided with access to the $\mathsf{PrivKey}$-oracle that returns the private key for the

given identity, which is computed in lines 5.2 and 5.3. The guard in line 5.1 denies queries for the challenge identity.

We would like to prove the probability judgment

$$[\, G^{\mathsf{BB}} : b = b' \,]_{\mathsf{adv}} \preceq [\, G_0^{\mathsf{DBDH}}[\mathsf{B} := B] : b \,]_{[\, G_1^{\mathsf{DBDH}}[\mathsf{B}:=B] \,:\, b \,]}$$

for some adversary definition B that can use A_1 and A_2. The judgment formalizes that the IND-sID-CPA advantage for all adversaries A_1 and A_2 instantiating A_1 and A_2 can be upper-bounded by the DBDH distinguishing probability for B (using A_1 and A_2).

In the next section, we will present a logic that allows us to prove the validity of such judgments $P \preceq P'$. The logic takes a probability expression P and during proof construction, the bound P' and the oracle polynomials and adversary definitions for reductions are synthesized. Our logic captures concrete security [11] since we explicitly relate adversaries and bounds in P and P'.

4. CORE LOGIC

In this section, we present our core logic. We focus on a subset of rules that is sufficient for proofs that do not require advanced techniques such as hybrid arguments or equivalence up to failure. We first present these core rules. Then, we formalize and prove soundness of the logic. Finally, we present algorithms for checking contextual equivalence of expressions. We will present high-level rules derived from the core rules in Section 5 and the remaining core rules in Section 6.

4.1 Core rules

Our logic relates probability judgments and consist of rules of the form

$$\frac{P_1 \preceq \epsilon_1 \quad \dots \quad P_k \preceq \epsilon_k}{P \preceq \epsilon}$$

where the P_i and P are probability expressions, the ϵ_i are metavariables, and ϵ is a probability expression built over the ϵ_i. The logic can be used to prove the validity of judgments of the form $[\, G : ev \,]_t \preceq \epsilon$ for a fixed ϵ or to synthesize ϵ while performing the proof. Our logic defines a small set of core rules which we then use to construct high-level rules that can be elaborated into core logic derivations.

For rules that perform contextual reasoning, we make use of the function $sp_p([\, G : ev \,])$ that computes the *strongest postcondition* at position p by collecting all (in)equalities in lets, random samplings, asserts, and guards on the path to p. The formal definition of *sp* can be found in Section 4.3. We now discuss the core rules given in Figure 5. The figure contains rules for dealing with probability judgments, bridging rules formalizing program transformations and equivalence of distributions, and rules for case distinctions and reductions.

Probability judgment rules

The Sym rule swaps the two security experiments in a distinguishing probability. This rule is required since all other rules only act on the first security experiment. The Deq rule formalizes that two identical games cannot be distinguished. The FalseEv rule formalizes that the probability of the event false is 0 in all games G. The Refl rule formalizes reflexivity. It is usually used if the premise is a hardness assumption that appears on the right-hand-side of the probability judgment to be proved. The Trans(SE') rule formalizes transitivity and is usually used in combination with other rules to derive new rules. For example, it is possible to derive inverses for all program transformation rules using Trans. The side condition enforces that no uninstantiated oracle or adversary symbols are introduced.

Program transformation rules

The Swap(p) rule swaps the command at positions p with the following command. Note that we require premise and conclusion to be well-formed, e.g., all variable occurences must be bound and bound variables must be distinct. This prevents Swap applications that lead to undefined variable occurences or change the semantics. The Add(p, c) rule inserts the command c at position p. As for Swap, the well-formenedness requirement is important since it prevents overshadowing of existing definitions. Add can be used together with Trans and Sym to remove commands. The Subst(p, e) rule performs contextual rewriting and replaces e' at position p by e if $sp_p(SE) \models e =_{\mathcal{E}} e'$ holds. The Subst rule is used for many different purposes in proofs. For example, it is often required to rewrite expressions before applying a cryptographic assumption, e.g., rewrite g^{a*b*x} to $(g^{a*b})^x$.

Random sampling and branching rules

The Rnd(p, t', C, C') rule formalizes optimistic sampling and replaces a uniform sampling from t by a uniform sampling from the distribution defined by $s \xleftarrow{\$} t'$; $\mathsf{return}\ C\{s\}$. To ensure that these distributions are equal, the rule checks that C is bijective in the given context using the provided inverse C'. The Except(p, b) rule adds the new excepted value b to the random sampling at position p. The bound is scaled by $\mathsf{n_o}$ if p is inside an oracle o. The CaseEv(c) rule performs a case distinction on the condition c in the event to bound both cases separately.

Reduction rules

The Abstract(G', B, B) rule moves some parts of the original game into a simulator B. In the arguments, G' is a game, B is an adversary symbol occuring exactly once in G', and the simulator $B = (B_{cs}, B_{ret})$ is a tuple of a sequence of commands and a return-expression. The simulator B can contain a hole to represent an input and oracle calls $y \leftarrow o(e)$ to oracles provided to B in G'. To apply the rule, the result of instantiating B with B in G' must be equal to G.

More formally, we define the (partial) function for adversary instantiation as follows. If B is not efficient or if any of the following steps fail, then $inst(G', \mathsf{B}, B) = \bot$. Otherwise, let $y, e, \vec{\mathsf{o}}, \vec{x}, \vec{c}, \vec{r}$ such that

$$y \leftarrow \mathsf{B}(e)\ \mathsf{with}\ \{\mathsf{o}_i(x_i) = \{c_i;\ \mathsf{return}\ r_i\}\}_{i=1}^{k}.$$

is the only call to B in G'. We denote the position of this call with p. Then $inst(G', \mathsf{B}, B) = G'\{B';\ \mathsf{let}\ y = B_{ret}\{e\}\}_p$ where B' is obtained from B_{cs} by plugging e into the hole and inlining all oracle calls.

Since rules should only introduce instantiated oracle and adversary symbols, the Abstract rule instantiates B with B and the new oracle symbols in G' with the polynomial bounds f_i. The bounds f_i are computed by inspecting B, e.g., if o_1 is called twice in the main body and once in an unbounded oracle o', then $f_1 = 2 + \mathsf{n}_{\mathsf{o}'}$.

Probability judgment:

$$\textsc{Sym}\ \frac{[\ SE'\ ]_{[SE]} \preceq \epsilon}{[\ SE\ ]_{[SE']} \preceq \epsilon} \qquad \textsc{Deq}\ \frac{}{[\ SE\ ]_{[SE]} \preceq 0} \qquad \textsc{FalseEv}\ \frac{}{[\ G : \mathsf{false}\ ]_t \preceq 0}\ \boxed{t \in \{\mathsf{succ}, \mathsf{adv}\}}$$

$$\textsc{Refl}\ \frac{}{[\ SE\ ]_t \preceq [\ SE\ ]_t} \qquad \textsc{Trans}(SE')\ \frac{[\ SE'\ ]_t \preceq \epsilon_1 \quad [\ SE\ ]_{[SE']} \preceq \epsilon_2}{[\ SE\ ]_t \preceq \epsilon_1 + \epsilon_2}\ \boxed{\begin{array}{l} asym(SE) = asym(SE') \text{ and} \\ osym(SE) = osym(SE') \end{array}}$$

Program transformation:

$$\textsc{Swap}(p)\ \frac{[\ SE\{c'; c\}_p\ ]_t \preceq \epsilon}{[\ SE\{c; c'\}_p\ ]_t \preceq \epsilon} \quad \textsc{Add}(p, c)\ \frac{[\ SE\{c; c'\}_p\ ]_t \preceq \epsilon}{[\ SE\{c'\}_p\ ]_t \preceq \epsilon}\ \boxed{\begin{array}{l} c \text{ sampling, let,} \\ \text{or } \mathsf{guard}(\mathsf{true}) \end{array}} \quad \textsc{Subst}(p, e)\ \frac{[\ SE\{e\}_p\ ]_t \preceq \epsilon}{[\ SE\{e'\}_p\ ]_t \preceq \epsilon}\ \boxed{sp_p(SE) \models e =_{\mathcal{E}} e'}$$

Random sampling:

$$\textsc{Rnd}(p, t', C, C')\ \frac{[\ SE\{s \xleftarrow{\$} t';\ \mathsf{let}\ r = C\{s\}\}_p\ ]_t \preceq \epsilon}{[\ SE\{r \xleftarrow{\$} t\}_p\ ]_t \preceq \epsilon}\ \boxed{sp_p(SE) \models C'\{C\} =_{\mathcal{E}} \square}$$

$$\textsc{Except}(p, b)\ \frac{[\ SE\{r \xleftarrow{\$} t \setminus b\}_p\ ]_t \preceq \epsilon}{[\ SE\{r \xleftarrow{\$} t\}_p\ ]_t \preceq \epsilon + s \times 1/|t|}\ \boxed{s = \begin{cases} \mathsf{n_o} \text{ if } p \text{ in } \mathsf{o} \\ 1 \quad \text{otherwise} \end{cases}}$$

Branching:

$$\textsc{CaseEv}(c)\ \frac{[\ G : ev \wedge c\ ]_t \preceq \epsilon_1 \quad [\ G : ev \wedge \neg c\ ]_t \preceq \epsilon_2}{[\ G : ev\ ]_t \preceq \epsilon_1 + \epsilon_2}\ \boxed{t \in \{\mathsf{succ}, \mathsf{adv}\}}$$

Reduction:

$$\textsc{Abstract}(G', \mathsf{B}, B)\ \frac{[\ G'[\mathsf{B} := B, \mathsf{n_{o_1}} := f_1, \ldots, \mathsf{n_{o_k}} := f_k] : ev\ ]_t \preceq \epsilon}{[\ G : ev\ ]_t \preceq \epsilon}\ \boxed{\begin{array}{l} G = inst(G', \mathsf{B}, B),\ asym(G') \subseteq asym(G) \uplus \{\mathsf{B}\}, \\ \mathsf{o_1}, \ldots, \mathsf{o_k} \text{ oracles provided to } \mathsf{B}, \\ \{\mathsf{o_1}, \ldots, \mathsf{o_k}\} \cap osym(G) = \emptyset, \text{ and} \\ f_i \text{ bounds for oracle queries to } o_i \text{ in } B \end{array}}$$

Figure 5: Core Rules. We require all security experiments in premises and conclusions to be well-formed.

4.2 Soundness of the core logic

In this section, we state and prove a soundness theorem for the core logic. For the proof, remember that our side-condition on rules ensures that all security experiments occuring in a derivation are well-formed.

THEOREM 1. *Let Δ denote a derivation of*

$$P \preceq P'$$

in the core logic, then $P \preceq P'$ is valid.

PROOF SKETCH. We perform a proof by induction over derivations. Let S denote an arbitrary P-setting. Let Δ_i denote derivations of $P_i \preceq P_i'$ for $i \in \{1, \ldots, k\}$ and assume Δ is the result of applying one of the core rules to the premises Δ_i. We then know that the premises $P_i \preceq P_i'$ are valid. For all rules except TRANS and ABSTRACT this implies that conditions (1) and (2) also hold for $P \preceq P'$. We proceed by performing a case distinction on the applied core rule:

- SYM, DEQ, FALSEEV, REFL: Immediately follows from the definition of validity and $prob_S$.
- TRANS(SE'): Conditions (1) and (2) for $[SE]_t \preceq \epsilon_1 + \epsilon_2$ are implied by the side-condition of the rule. To see that condition (3) holds, observe that the second premise yields $|\Pr_S[\ SE\ ] - \Pr_S[\ SE'\ ]| \leq \epsilon_2$.
- SWAP, ADD: Well-formedness of the premise implies that the distribution of the final memory (except for unused variables) coincides in both security experiments.
- SUBST: In the given context, evaluating e and e' yields the same result.
- RND: The distribution of the final memory is the same. Note that the variable s must be fresh since the premise is well-formed.
- EXCEPT(p, b): The games differ only if the value b is sampled. The probability of sampling b is reflected in the term $1/|t| \times s$.
- CASEEV: Follows from the fact that $\Pr_S[\ G : ev\ ] = \Pr_S[\ G : ev \wedge c\ ] + \Pr_S[\ G : ev \wedge \neg c\ ]$.
- ABSTRACT(G', B, B): The game G where B is inlined and the game G' where B is an instantiation argument are equivalent. $\square$

4.3 Checking contextual equivalence

For most of the rules presented so far, it is obvious how to implement proof checking if all rule arguments are explicitly given. The only exceptions are the rules SUBST and RND that both require a precise definition of $sp_p(SE)$ and algorithms for checking the conditional equivalence $\Gamma \models e =_{\mathcal{E}} e'$.

To define *sp*, we use the function *conseq* that takes a command c and returns a formula that characterizes its effect

on the state:

$$conseq(c) = \begin{cases} x \neq a & \text{if } c = x \xleftarrow{\$} t \setminus a \\ x = e & \text{if } c = \mathsf{let}\ x = e \\ b & \text{if } c = \mathsf{guard}(b) \\ nquant(ev) & \text{if } c = \mathsf{assert}(ev) \end{cases}$$

Here, $nquant(ev)$ denotes the (in)equalities in ev that are not below a quantifier. To compute $sp_p(G)$, we start with true and for each command c on the path to p, we add the conjunct $conseq(c)$ to the current post-condition. We ignore all oracle bodies unless p points into an oracle. If p points into an oracle, the commands in the oracle body preceding p are taken into account and all other oracle bodies are ignored. Since we can always rewrite the strongest post-condition into disjunctive normal form and check $\Gamma \models e =_{\mathcal{E}} e'$ for each disjunct separately, we assume w.l.o.g. that Γ is a conjunction of equalities and inequalities.

We use the following algorithm to rewrite e and e' into a normal-form and then check for syntactic equality. The algorithm combines separate algorithms for bitstrings, field expressions, and booleans that are applied bottom-up to values of the given types. In a first step, we simplify and orient the equalities in Γ and apply the resulting replacements to e and e'. Afterwards, values of type $\mathbb{G}_i$ are rewritten as g_i^f by using the log function and translating multiplication, division, and pairing to the corresponding operations on the exponents. Then we perform normalization bottom-up using the following approach:

- For bitstrings, we deal with $\oplus$ and 0 using the standard approach of flattening, sorting, filtering out 0, and counting occurences of expressions modulo 2.
- For values of type $\mathbb{F}_q$, our algorithm computes a normal-form of rational functions as used in computer algebra systems [29]. The algorithm represents ring expressions as polynomials over expressions e with non-field root symbols. It represents field expressions using a numerator polynomial f and a denominator polynomial h such that $\gcd(f, h) = 1$. Since the algorithm is only valid for well-defined expressions, we use a subroutine to prove $\bigwedge_{i=1}^{n} f_i \neq 0 \implies h \neq 0$ for all divisions by expressions h in e and e'. The subroutine uses gcd to find divisors h_j of the f_i such that $h = c * \prod_j h_j$ for some $c \in \mathbb{Q}$. Since we know $h_j \neq 0$, this implies $h \neq 0$. Concretely, the subroutine searches for i such that $h_j = gcd(h, f_i)$ nontrivial. If no such i exists, it fails. Otherwise it continues with $h := h/h_j$ until eventually, $h \in \mathbb{Q}$.
- For boolean values, we simplify equations by splitting equalities on product types, applying log to transform equalities on $\mathbb{G}_i$ into equalities on $\mathbb{F}_q$, and exploit the group structure for both bitstrings and field elements to obtain equations of the form $e = 0$. For field elements, e must be equal to $f * h^{-1}$ for normal-form ring expressions f and h. If e is well-defined, we further simplify $f * h^{-1} = 0$ to $f = 0$ and obtain a disjunction of inequalities $f_i = 0$ after factoring f into $\prod_i f_i$. For the logical operators, we apply the usual simplification rules such as $e \wedge e = e$ and $e \vee \mathsf{true} = \mathsf{true}$.
- For if-then-else, we detect common contexts C in both branches and simplify $b?C\{e\} : C\{e'\}$ to $C\{b?e : e'\}$.

Our implementation uses the Factory library packaged with the computer algebra system Singular [22] to perform the required computations on multivariate polynomials such as gcd or factoring. In the future, we might investigate the use of Groebner bases to check the satisfiability of sets of (in)equalities. But so far, we did not have problems with completeness using our approach.

5. HIGH LEVEL LOGIC

In theory, the core rules presented so far are sufficient to perform proofs that do not require hybrid arguments or equivalence up to failure. In practice, the level of abstraction is too low to perform non-trivial proofs.

In this section, we therefore derive high-level rules that capture standard arguments in cryptographic proofs. The high-level rules critically rely on algorithms that automatically infer arguments required for the elaboration into core rules. To perform fully automated proofs and to automatically discharge individual proof obligations, we then present a proof search procedure that finds derivations using our high-level rules.

5.1 Derived rules

The Simp rule unfolds all let-bindings and rewrites all expressions to their normal form. Simp is elaborated to applications of Subst that replace expressions by their normal form and applications of Trans and Add to remove (unused) let-bindings. The rule exploits that the normal form replaces all occurences of let-defined variables with their definitions. Optionally, the Simp rule can also use Swap to reorder commands in a unique, dependency-preserving way, remove exceptional values from samplings, and replace samplings of group elements by samplings of exponents.

The high-level Rnd* rule accepts a placeholder for one of C or C'. For example, given C and in a context where Γ holds, Rnd* uses a specialized algorithm for deducibility to find C' such that $\Gamma \models C, \vec{x} \vdash_{\mathcal{E}}^{C'} \square$ where $\vec{x}$ denotes all variables in G that are defined before the considered random sampling. Rnd* then uses the Simp rule to unfold the introduced let-binding. We will describe the algorithm for conditional deducibility in Section 5.3.

The Indep rule can be used to bound the probability of an event ev in G with $1/|t|$ if the event implies that G "guesses" the value of an unused randomly sampled variable. For all random variables r that are not used in the game, Indep tries to find an expression e not containing r such that $r = e$ is implied by ev. After adding such an equality to the event, Except can be used to sample r from $t \setminus e$. The event can then be simplified to false and we can conclude by applying FalseEv. To find such an e, Indep first combines all equalities in the event to obtain an equality of the form $(e_1, \ldots, e_k) = (0, \ldots, 0)$. Then, it uses a specialized algorithm for deducibility to find a C such that $\Gamma \models (e_1, \ldots, e_k), \vec{x} \vdash_{\mathcal{E}}^{C} r$ where $\vec{x}$ contains all variables that are defined before r is sampled. Since $C\{(e_1, \ldots, e_k)\} = C\{(0, \ldots, 0)\}$ is implied by the event and $C\{(0, \ldots, 0)\}$ does not contain r, it now suffices to exploit $\Gamma \models C\{(e_1, \ldots, e_k)\} =_{\mathcal{E}} r$ to get the desired equality of the form $r = e$.

The high-level rule IfEq replaces all occurences of $b?r_1 : r_2$ in a security experiment SE by r_1 if r_1 and r_2 are randomly sampled variables of type $\mathbb{F}_q$ that only occur in this context. The IfEq rule first applies Rnd* to replace r_2 by $r_1 + r_2$. This yields $r_1 + (b?0 : r_2)$ after simplification. Then, Rnd* is applied to the sampling of r_1 and replaces $r_1 + (b?0 : r_2)$ with r_1. This often makes the adversary's view independent of b and enables applications of Indep.

5.2 Automated application of assumptions

To apply a computational assumption such as $[\, G' : ev' \,]_{\mathsf{succ}}$ to a judgment $[\, G : ev \,]_{\mathsf{succ}}$, the ABSTRACT rule must be used followed by the REFL rule. To apply ABSTRACT, the simulator argument B has to be given explicitly and the instantiation of B in $G' : ev'$ with B has to be *syntactically equal* to $G : ev$. To apply an assumption, it is therefore necessary to perform the following steps:

1. Rewrite G and G' into a normal form, e.g., instead of sampling $X \in \mathbb{G}_i$, sample $x \in \mathbb{F}_q$ and compute g_i^x.
2. Swap and rename random samplings in G to match up with G'. The remainder of G will correspond to the simulator B.
3. Rewrite the part of G corresponding to B such that it does not use log or the random variables sampled by the challenger. To achieve this, the argument a used in the call $\mathsf{B}(a)$ in G' can be used instead. It might also be required to replace samplings of group elements in B by samplings of exponents.

We use the correct variant of SIMP to perform the first step unfolding all let-bindings in both games. If required to match up the samplings in both games, we also remove exceptions from samplings. Then, for all injections ρ from samplings in G' to samplings in G, we try the following. We match up the samplings according to ρ and match up adversary calls in G' with commands in G. To find the correct return value for the adversary call, we match up the two events ev and ev'. We now know the commands B'_{cs} and the return expression B'_{ret} that make G' equivalent to G. We still must rewrite these to (B_{cs}, B_{ret}) to satisfy the restrictions described in the third step. To achieve this, we use conditional deducibility and for each expression e in B' we try to find a C such that $\Gamma \models C\{a, \vec{x}\} =_{\mathcal{E}} e$ where a is the adversay input and $\vec{x}$ are the variables defined in B' before e is used.

The approach for applying a decisional assumption of the form $[\, G_0 : ev_0 \,]_{[\, G_1 \,:\, ev_1 \,]}$ is similar. Instead of discharging a judgment $[\, SE \,]_t$, it yields a new judgment $[\, SE' \,]_t$ where the difference between SE and SE' reflects the difference between $[\, G_0 : ev_0 \,]$ and $[\, G_1 : ev_1 \,]$. The high-level rule for decisional assumptions first rewrites SE such that it can be expressed as $inst(G_0, \mathsf{B}, B)$ for some B. Then it computes $SE' = inst(G_1, \mathsf{B}, B)$ and applies TRANS(SE'). The first premise of TRANS is $[\, SE' \,]_t$. The second premise is discharged by applying ABSTRACT (with the same B and B) to both $[SE]$ and $[SE']$ in the distinguishing probability followed by REFL.

5.3 Algorithms for conditional deducibility

So far, we have encountered three different high-level rules that require an algorithm to solve conditional deducibility problems. The RND* rule must find the inverse C' of a given context C. The INDEP rule must find a context C to extract a random variable from an expression. The rules for the automated application of assumptions must find (log-free) contexts C to deduce expressions e from known variables $\vec{x}$ and the adversary input a.

We have developed an algorithm following the approach described in [18] to deal with the combination of theories and to deal with Xor as a monoidal theory by solving equations over the associated semiring. For deducibility in groups and $\mathbb{F}_q$, we extended their approach to deal with the conditional axioms required to model inversion as a partial function. Our extension consists of two separate algorithms.

The first algorithm is used for RND* and INDEP and uses log to reduce deducibility in $\mathbb{G}_i$ to deducibility in $\mathbb{F}_q$. In both use-cases, the subroutine used for deducibility in $\mathbb{F}_q$ must solve problems of the form $\Gamma \models e, \vec{x} \vdash_{\mathcal{E}} y$ where $\vec{x}$ is a vector of variables of type $\mathbb{F}_q$, y is a variable of type $\mathbb{F}_q$, and e is a well-defined field-expression over the variables $\vec{x}$ and y. We can therefore normalize e and obtain two polynomials f and h such that e is equal to $f * h^{-1}$. If y occurs in both f and h, we give up. Otherwise, let $\{f, h\} = \{\hat{w}, w\}$ such that w contains y and $\hat{w}$ does not. We can then deduce $\hat{w}$ and it suffices to focus on deducing y from w. To achieve this, we try to find w_1, w_2 such that w is equal to $w_1 * y + w_2$ and w_i does not contain y and is hence deducible. If the degree of y in w is different from 1, this will not be possible and we give up. Otherwise, we can solve for y and since $\hat{w}$, w_1, and w_2 are deducible, we can deduce y from $f * h^{-1}$ which is either equal to $\hat{w} * (w_1 * y + w_2)^{-1}$ or to $(w_1 * y + w_2) * \hat{w}^{-1}$.

The second algorithm is more complicated since the context C cannot use log and we must distinguish between expressions in $\mathbb{F}_q$ that are known and expression that are only known "in the exponent". In the second case, we can only perform a limited number of multiplications using pairings and compute linear combinations using group multiplications. Our algorithm is tailored to problems of the form $\Gamma \models \vec{x}, g_{i_1}^{f_1}, \ldots, g_{i_k}^{f_k} \vdash_{\mathcal{E}} g_j^h$ where $\vec{x}$ is a vector of variables of type $\mathbb{F}_q$ and the f_i and h are polynomials. To solve such problems, we perform the following two steps keeping track of the context associated to each step.

1. Compute all group elements in $\mathbb{G}_j$ that can be obtained by applying pairings and isomorphisms to the given group elements $g_{i_u}^{f_u}$. This results in the new problem $\Gamma \models \vec{x}, g_j^{w_1}, \ldots, g_j^{w_l} \vdash_{\mathcal{E}} g_j^h$.
2. Search for polynomials u_i over $\vec{x}$ such that
$$\sum_{i=1}^{l} u_i * w_i = h.$$
To find such polynomials, we roughly proceed as follows. Perform a division with remainder of h by w_i to obtain u_i and b such that $h = u_i * w_i + b$. Check that u_i is a polynomial over $\vec{x}$ and continue with $h := b$. Since the division step might succeed with some w_j, but we might get stuck later on since there is no solution that uses w_j, we perform backtracking on the choice of divisors w_j.

5.4 Proof search

Between each step, our proof search procedure applies the simplification rule SIMP. This is critical since most steps are information-theoretical and exploit that group elements are always of the form $g_i^{f/h}$ for polynomials f and h. Next, we try to apply FALSEEV, INDEP, or a computational assumption that we want to use in the proof. Then, we try to make the view of an adversary A_i independent of random variables by applying RND* to replace contexts $C\{r\}$ occuring in the game by r. If this succeeds, then the other random variables occuring in C are not used in the given positions anymore. This might enable new applications of INDEP, e.g., if a variable r' that previously occcured in $C\{r\}$ in an adversary argument only occurs in the event afterwards. Another useful side-effect is that RND might remove products of vari-

$G^{\mathsf{BB},1} =$

1 : $i^* \leftarrow \mathsf{A}_1()$;
2 : $c, d, h, e \xleftarrow{\$} \mathbb{F}_q$; let $P = (g^c, g^d, g^h)$;
3 : $b \xleftarrow{\$} \mathbb{B}$; let $C = (g^e, g^{(d*i^*+h)*e})$;
4 : let $K_0 = \hat{\mathsf{e}}(g,g)^{c*d*e}$; $K_1 \xleftarrow{\$} \mathbb{G}_t$;
5 : $b' \leftarrow \mathsf{A}_2(P, C, (b?K_0 : K_1))$ with
PrivKey(i) = {
5.1 : guard$(i \neq i^*)$;
5.2 : $r \xleftarrow{\$} \mathbb{F}_q$;
5.3 : return $(g^{(c*d+r*(d*i+h))}, g^r)$
};

$G^{\mathsf{BB},2} =$

1 : $i^* \leftarrow \mathsf{A}_1()$;
2 : $c, d, h, e \xleftarrow{\$} \mathbb{F}_q$; let $P = (g^c, g^d, g^{h-d*i^*})$;
3 : $b \xleftarrow{\$} \mathbb{B}$; let $C = (g^e, g^{h*e})$;
4 : let $K_0 = \hat{\mathsf{e}}(g,g)^{c*d*e}$; $K_1 \xleftarrow{\$} \mathbb{G}_t$;
5 : $b' \leftarrow \mathsf{A}_2(P, C, (b?K_0 : K_1))$ with
PrivKey(i) = {
5.1 : guard$(i \neq i^*)$;
5.2 : $r \xleftarrow{\$} \mathbb{F}_q$;
5.3 : return $(g^{(c*d+r*(d*(i-i^*)+h))}, g^r)$
};

$G^{\mathsf{BB},4} =$

1 : $i^* \leftarrow \mathsf{A}_1()$;
2 : $c, d, h, e, t \xleftarrow{\$} \mathbb{F}_q$; let $P = (g^c, g^d, g^{h-d*i^*})$;
3 : $b \xleftarrow{\$} \mathbb{B}$; let $C = (g^e, g^{h*e})$;
4 : let $K_0 = \hat{\mathsf{e}}(g,g)^t$; $K_1 \xleftarrow{\$} \mathbb{G}_t$;
5 : $b' \leftarrow \mathsf{A}_2(P, C, (b?K_0 : K_1))$ with
PrivKey(i) = {
5.1 : guard$(i \neq i^*)$;
5.2 : $r \xleftarrow{\$} \mathbb{F}_q$;
5.3 : return $(g^{(c*d+r*(d*(i-i^*)+h))}, g^r)$
};

$G^{\mathsf{BB},5} =$

1 : $i^* \leftarrow \mathsf{A}_1()$;
2 : $c, d, h, e, t \xleftarrow{\$} \mathbb{F}_q$; let $P = (g^c, g^d, g^{h-d*i^*})$;
3 : $b \xleftarrow{\$} \mathbb{B}$; let $C = (g^e, g^{h*e})$;
5 : $b' \leftarrow \mathsf{A}_2(P, C, \hat{\mathsf{e}}(g,g)^t))$ with
PrivKey(i) = {
5.1 : guard$(i \neq i^*)$;
5.2 : $r \xleftarrow{\$} \mathbb{F}_q$;
5.3 : return $(g^{(c*d+r*(d*(i-i^*)+h))}, g^r)$
};

Figure 6: Proof of the Boneh-Boyen IBKEM using our high-level rules.

ables that are not deducible from the values provided by the challenger for an assumption. Finally, we try to to apply one of the decisional assumptions specified in the given context. To exclude useless applications of decisional assumptions, we check that the adversary arguments that differ in G_0 and G_1 are used. To prevent cycles, we also disallow applications of assumptions that undo previous applications by applying the assumption in the opposite direction. In general, non-termination might still be possible and we therefore bound the size of the explored proof trees.

Example 2. We now prove the judgment from Example 1 using the high-level rules. Our implementation of the proof search automatically finds the following proof.

The proof search first applies Simp which yields the game $G^{\mathsf{BB},1}$ given in Figure 6. We keep the let-definitions in our presentation to increase readability.

In the next step, the proof search applies the Rnd* rule to the sampling of h. The proof search discovers that h is used in the context $h + d * i^*$ in the exponent of C_2 and that replacing $h + d * i^*$ by h removes all occurences of the product $d * e$ from the game. The elaborated core rule application is

$$\text{Rnd}(p_h, \Box - d * i^*, \Box + d * i^*)$$

where p_h is the position of the sampling of h. The rule replaces $h \xleftarrow{\$} \mathbb{F}_q$ by $h' \xleftarrow{\$} \mathbb{F}_q$; let $h = h' - d * i^*$ and after unfolding, the new exponent of C_2 is $(d*i^* + (h' - d*i^*))*e = h' * e$. After renaming h' to h, we get game $G^{\mathsf{BB},2}$ given in Figure 6.

Next, the proof search focuses on the product $c * d$ in the exponent of the first group element returned by PrivKey and applies Rnd* to replace the expression $r * (i - i^*) + c$ by r The actual core rule application is

$$\text{Rnd}(p_r, (\Box - c)/(i - i^*), \Box * (i - i^*) + c)$$

where p_r is the position of the sampling of r in the oracle. Here, $sp_{p_r}(G^{\mathsf{BB},2})$ includes the axiom $i \neq i^*$ which is required to prove that the second context is the inverse of the first. After simplification, line 5.3 changes to

$$5.3 : \quad \text{return } (g^{d*r+h*(r-c)/(i-i^*)}, g^{(r-d)/(i-i^*)}).$$

In the next step, the proof search automatically applies the DBDH assumption. The rule matches up the samplings of c, d, e in the game with the samplings of a, b, c in the DBDH assumption. The rule synthesizes the simulator given in Figure 7 using $(\xi_1, \xi_2, \xi_3, \xi_4) = (g^c, g^d, g^e, \hat{\mathsf{e}}(g,g)^{c*d*e})$ to denote B's input. Applying the rule results in the game $G^{\mathsf{BB},4}$ given in Figure 6. The game $G^{\mathsf{BB},4}$ differs from $G^{\mathsf{BB},3}$ in the sampling of t and the value assigned to K_0.

To finish the proof, the proof search first applies IfEq to obtain game $G^{\mathsf{BB},5}$ given in Figure 6. Afterwards b does not occur anywhere except in the event $b = b'$ and the proof is concluded by applying the Indep rule to the randomly sampled boolean b. The application of Indep yields the desired probability bound $\frac{1}{2}$.

6. ADVANCED CORES RULES

The set of basic core rules are complemented by a set of advanced core rules which are required for more advanced examples. The set of advanced core rules is given in Figure 8. The set contains rules new rules for program transformations, equivalence up to failure, and hybrid arguments.

Program transformation:

$$\textsc{Assert}(c)\ \frac{[\ G;\ \mathsf{assert}(c) : ev \wedge c\ ]_t \preceq \epsilon}{[\ G : ev \wedge c\ ]_t \preceq \epsilon}$$

Equivalence up to failure:

$$\textsc{Upto}(p,c)\ \frac{[\ G\{\mathsf{guard}(c)\}_p : ev\ ]_t \preceq \epsilon_1 \qquad [\ G\{\mathsf{guard}(c)\}_p : \exists\, x \in Q_o.\, c(x) \neq c'(x)\ ]_{\mathsf{succ}} \preceq \epsilon_2}{[\ G\{\mathsf{guard}(c')\}_p : ev\ ]_t \preceq \epsilon_1 + \epsilon_2}\ \boxed{p \text{ first position in } o}$$

$$\textsc{Guess}\ \frac{[\ G;\ x \leftarrow \mathsf{A}() : ev\ ]_t \preceq \epsilon \times \mathsf{n}_o}{[\ G : \exists\, x \in Q_o.\, ev\ ]_t \preceq \epsilon} \qquad \textsc{Find}(C,e)\ \frac{[\ G;\ x \leftarrow \mathsf{A}(e) : ev_1 \wedge ev_2\ ]_t \preceq \epsilon}{[\ G : (\exists\, x \in Q_o.\, ev_1) \wedge ev_2\ ]_t \preceq \epsilon}\ \boxed{\begin{array}{l} C \text{ efficient and} \\ sp_{|G|}(G) \models C\{(e,x)\} =_{\mathcal{E}} ev_1 \end{array}}$$

Hybrid Arguments:

$$\textsc{Hybrid}(p,ob)\ \frac{[\ G\{ob\}_p : ev\ ]_t \preceq \epsilon_1 \quad [\ SE_1\ ]_{[SE_2]} \preceq \epsilon_2}{[\ G\{ob'\}_p : ev\ ]_t \preceq \epsilon_1 + \mathsf{n}_o \times \epsilon_2}\ \boxed{\begin{array}{l} p \text{ points to body of } o,\ ev' = splitQuants(o, ev), \\ SE_1 = [\ G\{\mathsf{bif}^{<} : ob \mid \mathsf{bif}^{=} : ob' \mid \mathsf{bif}^{>} : ob'\}_p : ev'\ ], \text{ and} \\ SE_2 = [\ G\{\mathsf{bif}^{<} : ob \mid \mathsf{bif}^{=} : ob \mid \mathsf{bif}^{>} : ob'\}_p : ev'\ ] \end{array}}$$

$$\textsc{OSwap}(p)\ \frac{[\ SE\{\}_p\{x \xleftarrow{\$} \mathcal{D}\}_{p'}\ ]_t \preceq \epsilon}{[\ SE\{x \xleftarrow{\$} \mathcal{D}\}_p\ ]_t \preceq \epsilon}\ \boxed{\begin{array}{l} p \text{ first position in } \mathsf{bif}^{=} \text{ of hybrid oracle} \\ p' \text{ position before adversary call containing } p \end{array}}$$

Figure 8: Advanced Core Rules.

$$B = \left|\begin{array}{ll} 1: & i^* \leftarrow \mathsf{A}_1(); \\ 2: & h \xleftarrow{\$} \mathbb{F}_q; \mathsf{let}\ P = (\xi_1, \xi_2, \xi_2^{i^*} * g^h); \\ 3: & b \xleftarrow{\$} \mathbb{B};\ \mathsf{let}\ C = (\xi_3, \xi_3^h; \\ 4: & \mathsf{let}\ K_0 = \xi_4;\ K_1 \xleftarrow{\$} \mathbb{G}_t; \\ 5: & b' \leftarrow \mathsf{A}_2(P, C, (b ? K_0 : K_1))\ \mathsf{with} \\ & \quad \mathsf{PrivKey}(i) = \{ \\ 5.1: & \quad \mathsf{guard}(i \neq i^*); \\ 5.2: & \quad r \xleftarrow{\$} \mathbb{F}_q; \\ 5.3: & \quad \mathsf{return}\ \ (\xi_2^r * g^{h*r/(i-i^*)} * \xi_1^{-h/(i-i^*)} \\ & \qquad\qquad , g^{r/(i-i^*)} * \xi_2^{-1/(i-i^*)}) \\ & \quad \}; \\ 6: & \mathsf{return}\ \ b = b' \end{array}\right|$$

Figure 7: Synthesized simulator for DBDH.

6.1 Program transformation

The $\textsc{Assert}(c)$ rule appends the command $\mathsf{assert}(c)$ to the body of the game and requires that the event already contains the conjunct c. The $\mathsf{assert}(c)$ can then be moved further up using Swap if c is well-defined at the given position. Then the condition c can be exploited to simplify later commands.

6.2 Equivalence up to failure

The $\textsc{Upto}(p,c)$ rule replaces $\mathsf{guard}(c')$ at position p in an oracle with $\mathsf{guard}(c)$ and yields two proof obligations: In the resulting game, bound the probability of the original event and the probability that the adversary performs a query where the results of c and c' differ. To deal with the existential quantifiers introduced by Upto, the following rules are used. The Guess and $\textsc{Find}(C,e)$ rules get rid of an existential quantification $\exists x \in Q_o.\, ev$ in the event by introducing an adversary that guesses or finds an $e \in Q_o$ for which ev is true. The arguments e and C of the Find rule define the argument given to A and (efficient) test executed by A to determine if a given $x \in Q_o$ satisfies ev.

6.3 Hybrid arguments

To formalize hybrid arguments, we first extend the syntax and semantics of games with hybrid oracles. A hybrid oracle has a body of the following form:

$$[\mathsf{bif}^{<} : ob_1 \mid \mathsf{bif}^{=} : ob_2 \mid \mathsf{bif}^{>} : ob_3]$$

Here, ob_1, ob_2, and ob_3 are ordinary oracle bodies consisting of a sequence of oracle commands and a return expression. To execute a game containing hybrid oracles, a value i_o in $\{0, \ldots, \delta_o - 1\}$ is sampled initially for each hybrid oracle o. The oracle body for a hybrid oracle o is defined as

$$\begin{array}{l} \mathsf{if}\ \ (c_o < i_o)\ \mathsf{then}\ \mathsf{bif}^{<} \\ \mathsf{elif}\ (c_o = i_o)\ \mathsf{then}\ \mathsf{bif}^{=} \\ \mathsf{elif}\ (c_o > i_o)\ \mathsf{then}\ \mathsf{bif}^{>}. \end{array}$$

For Hybrid oracles, the query loq Q_o contains only arguments for queries with $c_o \neq i_o$ and the argument of the i_o-query is stored in a global variable.

Hybrid oracles are required to express the proof obligations of the Hybrid rule. The $\textsc{Hybrid}(p, ob)$ rule replaces the oracle body ob' with ob at position p and yields two proof obligations: Bound the original event ev in the resulting game and bound the distinguishing probability for the two hybrid games. In both hybrid games, ob is used if $c_o < i_o$ and ob' is used if $c_o > i_o$. If $c_o = i_o$, the first hybrid game uses ob' and the second hybrid game uses ob. The modified event ev' accounts for the fact that the query log does not contain the argument for the i_o-th query anymore. It uses the function *splitQuants* to replace quantifications such as $\forall x \in Q_o.\, c \neq c^*$ with $(\forall x \in Q_o.\, c \neq c^*) \wedge c \neq c^*$ where c refers to the argument of the i_o-th query.

A related rule is the OSwap(p) rule that requires the position p to point into a hybrid oracle definition. More precisely, p must point to the first command of the body for $i_o = c_o$, and this must be a sampling. OSwap exploits that the body for $i_o = c_o$ is executed at most once and moves this sampling to the main body of the game immediately before the adversary call containing the oracle definition.

7. TOOL AND CASE STUDIES

We have implemented the logic and the described algorithms in the AutoG&P tool and verified its effectiveness on the case studies presented in Table 1.[1] The source code of AutoG&P comprises about 13K lines of OCaml with about 3KLoC each for proof search and extraction to EasyCrypt. The tool performs proof search with a bound on the size of the proof tree to ensure termination.

7.1 Case studies

The first four entries of the table are smaller examples that can be proven automatically except for the first one. The first example proves the implication between two assumptions and requires a creative step that the proof search (expectedly) does not discover.

The Cramer-Shoup encryption scheme and the Kurosawa-Desmedt encryption scheme use cyclic groups and are IND-CCA secure. Our proofs of the two schemes closely follow the published proofs and yield similar bounds. For Cramer-Shoup, we provide two proofs. The first proof is manual (25 lines) and checked in one second. The second proof is discovered fully automatically by the proof search algorithm. The proof is essentially identical to our manual proof and is found in around 12 seconds. The structure-preserving encryption scheme by Camenisch et al. uses bilinear groups of Type I. Again, the structure of our proofs closely follows the pen-and-paper proof. All three proofs rely on the Upto rule for reasoning up to failure.

The proof of selective security for the Boneh-Boyen IBE scheme is discovered automatically both for the Type I and Type III versions of the scheme. Somewhat surprisingly, the sequence of high-level rule applications is identical for both settings which is promising for certified automated translation between settings. The proof of full security for the Water dual-system IBE follows the dual-system methodology, i.e., first the challenge cipher-text is encrypted using the so-called semi-functional encryption algorithm, then a hybrid argument is used to replace the key generation algorithm by a semi-functional version, and finally, it is shown that the view of the adversary is independent of the bit b.

7.2 EasyCrypt proof generation

We have implemented a proof generation mechanism that transforms a valid derivation in our logic into a file that can be verified independently using EasyCrypt. Generation is done in four steps:

1. build a context that declares all size variables, operators, constants and global variables required in the different games of the proof. This step translates the signature and the setting into EasyCrypt;

2. build the sequence of games, including the code of the simulators in reduction steps;

3. output judgments in the relational and ambient logics of EasyCrypt to justify all steps in the derivation tree. This step critically uses high-level proof principles formalized and proved in EasyCrypt libraries;

4. prove the concluding claim by combining all previous derived inequalities.

The generation algorithm involves some non-trivial "plumbing" between the two systems. In the long term, we plan to enhance automation in EasyCrypt by providing a tighter integration of AutoG&P and EasyCrypt.

[1] The AutoG&P tool and the case studies are available at `https://github.com/ZooCrypt/AutoGnP`.

8. RELATED WORK

There is an increasing number of tools for proving the security of cryptographic constructions in the computational model. The oldest tool is CryptoVerif [13], which has been used for protocols and a few primitives. To our best knowledge, CryptoVerif achieves best automation for protocols and has never been used to verify pairing-based constructions. More recent tools, such as CertiCrypt [9], EasyCrypt [8], and FCF [34] can be used to reason about protocols and primitives. Indeed, CertiCrypt has been used to verify the chosen plaintext security of Boneh and Franklin Identity Based Encryption in the random oracle model [10]. However, these tools are mostly interactive and proofs are very long and can only be built by experts. On the other hand, there exist specialized tools, such as [19], [27], [26], [33], [31] and [7], which achieve complete automation for specialized classes of constructions (padding-based encryption, message authentication codes, modes of operation, authenticated encryption, bounded security of structure-preserving signatures in the generic group model). Our work is closely related to ZooCrypt [5], which relies on a powerful domain-specific logic to reason about chosen-plaintext and chosen-ciphertext security of padding-based encryption schemes; in particular, our work generalizes the idea of algebraic reduction from ZooCrypt. However, our logic is applicable to a broad range of constructions—although AutoG&P is focused on group-based and pairing-based cryptography. Moreover, our core logic has strong connections with Computational Indistinguishability Logic, or CIL [6], a general-purpose logic to reason about security of cryptographic constructions. However, CIL does not provide a syntax for describing games and does not offer support for automation.

9. CONCLUSION

We have introduced a formal logic that supports concise and intuitive proofs of cryptographic constructions, and presented the AutoG&P tool, which implements the logic for the specialized case of pairing-based cryptography. Our experiments show that formal proofs of complex pairing-based constructions are now within reach. Future work includes extending the scope of AutoG&P to accommodate i. q-type and interactive assumptions; ii. random oracles; iii. key exchange protocols; iv. other types of constructions, including (structure preserving) signatures; v. other areas, including lattice-based cryptography. In addition, we plan to explore the possibility to build on top of AutoG&P automated proof transformations from Type I to Type III settings [4, 3], or composite-order via prime-order groups [25, 32, 30].

Case study			Proof	
Reference	Scheme	Property	LoC	Time (s)
Abe et al. '10 [1]	DDH ⇒ DP assumption	reduction	4	1
ElGamal '84 [23]	ElGamal encryption	IND-CPA	auto	1
Escala et al. '13 [24]	Matrix D-Lin Encryption	IND-CPA	auto	1
Escala et al. '13 [24]	Matrix S-Casc Encryption	IND-CPA	auto	1
Cramer and Shoup '98 [20]	Cramer-Shoup encryption	IND-CCA	25/auto	1/12
Abe et al. '05 [2]	Kurosawa-Desmedt encryption	IND-CCA	70	2
Camenisch et al. '11 [16]	Structure-preserving encryption	IND-CCA	22	12
Boneh and Boyen '04 [14]	Boneh-Boyen IBE	sID-IND-CPA	auto	2
Waters '09 [36]	Waters dual-system IBE	ID-IND-CPA	98	3

Table 1: Case studies

Acknowledgement

This work is supported in part by ONR grant N00014-12-1-0914, Madrid regional project S2009TIC-1465 PROMETIDOS, and Spanish national projects TIN2009-14599 DESAFIOS 10, and TIN2012-39391-C04-01 Strongsoft. The research of Schmidt has received funds from the European Commission's Seventh Framework Programme Marie Curie Cofund Action AMAROUT II (grant no. 291803).

10. REFERENCES

[1] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. Structure-preserving signatures and commitments to group elements. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 209–236. Springer, Aug. 2010.

[2] M. Abe, R. Gennaro, K. Kurosawa, and V. Shoup. Tag-KEM/DEM: A new framework for hybrid encryption and a new analysis of Kurosawa-Desmedt KEM. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 128–146. Springer, May 2005.

[3] M. Abe, J. Groth, M. Ohkubo, and T. Tango. Converting cryptographic schemes from symmetric to asymmetric bilinear groups. In *Advances in Cryptology – CRYPTO 2014, Part I*, Lecture Notes in Computer Science, pages 241–260. Springer, Aug. 2014.

[4] J. A. Akinyele, M. Green, and S. Hohenberger. Using SMT solvers to automate design tasks for encryption and signature schemes. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 399–410. ACM Press, Nov. 2013.

[5] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Z. Béguelin. Fully automated analysis of padding-based encryption in the computational model. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 1247–1260. ACM Press, Nov. 2013.

[6] G. Barthe, M. Daubignard, B. M. Kapron, and Y. Lakhnech. Computational indistinguishability logic. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 375–386. ACM Press, Oct. 2010.

[7] G. Barthe, E. Fagerholm, D. Fiore, A. Scedrov, B. Schmidt, and M. Tibouchi. Strongly-optimal structure preserving signatures from type II pairings: Synthesis and lower bounds. In J. Katz, editor, *Public-Key Cryptography - PKC 2015*, volume 9020 of *Lecture Notes in Computer Science*, pages 355–376. Springer, 2015.

[8] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, Aug. 2011.

[9] G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101, New York, 2009. ACM.

[10] G. Barthe, F. Olmedo, and S. Z. Béguelin. Verifiable security of boneh-franklin identity-based encryption. In *Proceedings of the 5th International Conference on Provable Security*, ProvSec'11, pages 68–83, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 394–403. IEEE, 1997.

[12] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 82. IEEE Computer Society, 2001.

[13] B. Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society Press, May 2006.

[14] D. Boneh and X. Boyen. Efficient selective-ID secure identity based encryption without random oracles. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 223–238. Springer, May 2004.

[15] D. Boneh and A. Silverberg. Applications of multilinear forms to cryptography. *Contemporary Mathematics*, 324(1):71–90, 2003.

[16] J. Camenisch, K. Haralambiev, M. Kohlweiss, J. Lapon, and V. Naessens. Structure preserving CCA secure encryption and applications. In D. H. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 89–106. Springer, Dec. 2011.

[17] J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical multilinear maps over the integers. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 476–493. Springer, Aug. 2013.

[18] V. Cortier and S. Delaune. Decidability and combination results for two notions of knowledge in security protocols. *Journal of Automated Reasoning*, 48(4):441–487, 2012.

[19] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM CCS 08: 15th Conference on Computer and Communications Security*, pages 371–380. ACM Press, Oct. 2008.

[20] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25. Springer, Aug. 1998.

[21] C. J. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification*, pages 414–418. Springer, 2008.

[22] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 4-0-2 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2015.

[23] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, Aug. 1984.

[24] A. Escala, G. Herold, E. Kiltz, C. Ràfols, and J. Villar. An algebraic framework for Diffie-Hellman assumptions. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 129–147. Springer, Aug. 2013.

[25] D. M. Freeman. Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In H. Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 44–61. Springer, May 2010.

[26] M. Gagné, P. Lafourcade, and Y. Lakhnech. Automated security proofs for almost-universal hash for MAC verification. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 291–308. Springer, Sept. 2003.

[27] M. Gagné, P. Lafourcade, Y. Lakhnech, and R. Safavi-Naini. Automated security proof for symmetric encryption modes. In A. Datta, editor, *Advances in Computer Science - ASIAN 2009*, volume 5913 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2009.

[28] S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17. Springer, May 2013.

[29] K. O. Geddes and G. Labahn. *Algorithms for computer algebra*. Springer Science & Business Media, 1992.

[30] G. Herold, J. Hesse, D. Hofheinz, C. Ràfols, and A. Rupp. Polynomial spaces: A new framework for composite-to-prime-order transformations. In *Advances in Cryptology – CRYPTO 2014, Part I*, Lecture Notes in Computer Science, pages 261–279. Springer, Aug. 2014.

[31] V. T. Hoang, J. Katz, and A. J. Malozemoff. Automated analysis and synthesis of authenticated encryption schemes. In *ACM Conference on Computer and Communications Security*, 2015.

[32] A. B. Lewko. Tools for simulating features of composite order bilinear groups in the prime order setting. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 318–335. Springer, Apr. 2012.

[33] A. J. Malozemoff, J. Katz, and M. D. Green. Automated analysis and synthesis of block-cipher modes of operation. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 140–152. IEEE, 2014.

[34] A. Petcher and G. Morrisett. The foundational cryptography framework. In R. Focardi and A. C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST*, volume 9036 of *Lecture Notes in Computer Science*, pages 53–72. Springer, 2015.

[35] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012.

[36] B. Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In S. Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 619–636. Springer, Aug. 2009.