The Hidden Number Problem

Sage Binder, Eric Ren, and Katherine Kosaian

September 1, 2025

Abstract

In this entry, we formalize the Hidden Number Problem (HNP), originally introduced by Boneh and Venkatesan in 1996 [2]. Intuitively, the HNP involves demonstrating the existence of an algorithm (the "adversary") which can compute (with high probability) a hidden number α given access to a bit-leaking oracle. Originally developed to establish the security of Diffie–Hellman key exchange, the HNP has since been used not only for protocol security but also in cryptographic attacks, including notable ones on DSA and ECDSA.

Additionally, the HNP makes use of an instance of Babai's nearest plane algorithm [1], which solves the approximate closest vector problem. Thus, building on the LLL algorithm [5] (which has already been formalized [4, 3, 6]), we formalize Babai's algorithm, which itself is of independent interest. Our formalizations of Babai's algorithm and the HNP adversary are executable, setting up potential future work, e.g. in developing formally verified instances of cryptographic attacks.

Note, our formalization of Babai's algorithm here is an updated version of a previous AFP entry of ours. The updates include a tighter error bound, which is required for our HNP proof.

Contents

theory *Misc-PMF*
imports
HOL−Probability.Probability
LLL-Basis-Reduction.LLL-Impl

begin

definition *replicate-spmf* :: *nat* ⇒ *'b pmf* ⇒ *'b list spmf* **where**
replicate-spmf m p = *spmf-of-pmf* (*replicate-pmf m p*)

The preceding *replicate-spmf* definition is copied from *CRYSTALS−Kyber-Security*. We do this to avoid loading the entire library. In fact, we do not need *replicate-spmf* at all for the HNP. However, for each *replicate-pmf* result we prove here, we also prove a corresponding *replicate-spmf* result. The *replicate-spmf* results are here for completeness, but not needed for the HNP.

1 SPMF and PMF helper lemmas

lemma *spmf-eq-element*: *spmf* (*p* ⤜ (λ*x*. *return-spmf* (*x* = *t*))) *True* = *spmf p t*
⟨*proof*⟩

lemma *pmf-true-false*:
fixes *p* :: *'a pmf*
fixes *P Q* :: *'a* ⇒ *bool*
defines *a* ≡ *p* ⤜ (λ*x*. *return-pmf* (*P x*))
defines *b* ≡ *p* ⤜ (λ*x*. *return-pmf* (¬ *P x*))
shows *pmf a True* = *pmf b False*
⟨*proof*⟩

lemma *spmf-true-false*:
fixes *p* :: *'a spmf*
fixes *P Q* :: *'a* ⇒ *bool*
defines *a* ≡ *p* ⤜ (λ*x*. *return-spmf* (*P x*))

defines $b \equiv p \gg= (\lambda x.\ \textit{return-spmf}\ (\neg\ P\ x))$
shows $\textit{spmf}\ a\ \textit{True} = \textit{spmf}\ b\ \textit{False}$
⟨*proof*⟩

lemma *pmf-subset*:
fixes $p :: {'a}\ \textit{pmf}$
fixes $P\ Q :: {'a} \Rightarrow \textit{bool}$
assumes $\forall\, x \in \textit{set-pmf}\ p.\ P\ x \longrightarrow Q\ x$
shows $\textit{pmf}\ (p \gg= (\lambda x.\ \textit{return-pmf}\ (P\ x)))\ \textit{True} \le \textit{pmf}\ (p \gg= (\lambda x.\ \textit{return-pmf}\ (Q\ x)))\ \textit{True}$
⟨*proof*⟩

lemma *pmf-subset′*:
fixes $p :: {'a}\ \textit{pmf}$
fixes $P\ Q :: {'a} \Rightarrow \textit{bool}$
assumes $\forall\, x \in \textit{set-pmf}\ p.\ \neg\ P\ x \longrightarrow \neg\ Q\ x$
shows $\textit{pmf}\ (p \gg= (\lambda x.\ \textit{return-pmf}\ (P\ x)))\ \textit{False} \le \textit{pmf}\ (p \gg= (\lambda x.\ \textit{return-pmf}\ (Q\ x)))\ \textit{False}$
⟨*proof*⟩

lemma *spmf-subset*:
fixes $p :: {'a}\ \textit{spmf}$
fixes $P\ Q :: {'a} \Rightarrow \textit{bool}$
assumes $\forall\, x \in \textit{set-spmf}\ p.\ P\ x \longrightarrow Q\ x$
shows $\textit{spmf}\ (p \gg= (\lambda x.\ \textit{return-spmf}\ (P\ x)))\ \textit{True} \le \textit{spmf}\ (p \gg= (\lambda x.\ \textit{return-spmf}\ (Q\ x)))\ \textit{True}$
⟨*proof*⟩

lemma *spmf-subset′*:
fixes $p :: {'a}\ \textit{spmf}$
fixes $P\ Q :: {'a} \Rightarrow \textit{bool}$
assumes $\forall\, x \in \textit{set-spmf}\ p.\ \neg\ P\ x \longrightarrow \neg\ Q\ x$
shows $\textit{spmf}\ (p \gg= (\lambda x.\ \textit{return-spmf}\ (P\ x)))\ \textit{False} \le \textit{spmf}\ (p \gg= (\lambda x.\ \textit{return-spmf}\ (Q\ x)))\ \textit{False}$
⟨*proof*⟩

lemma *pmf-in-set*:
fixes $A :: {'a}\ \textit{set}$
fixes $p :: {'a}\ \textit{pmf}$
shows $\textit{pmf}\ (p \gg= (\lambda x.\ \textit{return-pmf}\ (x \in A)))\ \textit{True} = \textit{prob-space.prob}\ p\ A$
⟨*proof*⟩

lemma *pmf-of-prop*:
fixes $P :: {'a} \Rightarrow \textit{bool}$
fixes $p :: {'a}\ \textit{pmf}$
shows $\textit{pmf}\ (p \gg= (\lambda x.\ \textit{return-pmf}\ (P\ x)))\ \textit{True} = \textit{prob-space.prob}\ p\ \{x \in \textit{set-pmf}\ p.\ P\ x\}$
⟨*proof*⟩

lemma *spmf-in-set*:
fixes $A :: 'a\ set$
fixes $p :: 'a\ spmf$
shows $spmf\ (p \gg= (\lambda x.\ return\text{-}spmf\ (x \in A)))\ True = prob\text{-}space.prob\ p\ (Some`A)$
$\langle proof \rangle$

lemma *spmf-of-prop*:
fixes $P :: 'a \Rightarrow bool$
fixes $p :: 'a\ spmf$
shows $spmf\ (p \gg= (\lambda x.\ return\text{-}spmf\ (P\ x)))\ True = prob\text{-}space.prob\ p\ (Some`\{x \in set\text{-}spmf\ p.\ P\ x\})$
$\langle proof \rangle$

lemma *replicate-pmf-events-helper*:
fixes $p :: 'a\ pmf$
fixes $n\ P$
defines $lhs \equiv pmf\ (pair\text{-}pmf\ p\ (replicate\text{-}pmf\ n\ p) \gg= (\lambda(x,\ xs).\ return\text{-}pmf\ (x\ \#\ xs)) \gg= (\lambda xs.\ return\text{-}pmf\ (P\ xs)))\ True$
defines $rhs \equiv pmf\ (pair\text{-}pmf\ p\ (replicate\text{-}pmf\ n\ p) \gg= (\lambda(x,\ xs).\ return\text{-}pmf\ (P\ (x\ \#\ xs))))\ True$
shows $lhs = rhs$
$\langle proof \rangle$

lemma *replicate-pmf-events*:
fixes $p :: 'a\ pmf$
fixes $n :: nat$
fixes $E :: nat \Rightarrow 'a \Rightarrow bool$
assumes $\forall\, i < n.\ pmf\ (p \gg= (\lambda x.\ return\text{-}pmf\ (E\ i\ x)))\ True = A\ i$
shows $pmf\ (replicate\text{-}pmf\ n\ p \gg= (\lambda xs.\ return\text{-}pmf\ (\forall\, i < n.\ E\ i\ (xs!i))))\ True = (\prod i{<}n.\ A\ i)$
$\langle proof \rangle$

lemma *replicate-pmf-same-event*:
fixes $p :: 'a\ pmf$
fixes $n :: nat$
fixes $E :: 'a \Rightarrow bool$
assumes $pmf\ (p \gg= (\lambda x.\ return\text{-}pmf\ (E\ x)))\ True = A$
shows $pmf\ (replicate\text{-}pmf\ n\ p \gg= (\lambda xs.\ return\text{-}pmf\ (\forall\, i < n.\ E\ (xs!i))))\ True = A\widehat{\ }n$
$\langle proof \rangle$

lemma *replicate-pmf-same-event-leq*:
fixes $p :: 'a\ pmf$
fixes $n :: nat$
fixes $E :: 'a \Rightarrow bool$
assumes $pmf\ (p \gg= (\lambda x.\ return\text{-}pmf\ (E\ x)))\ True \le A$
shows $pmf\ (replicate\text{-}pmf\ n\ p \gg= (\lambda xs.\ return\text{-}pmf\ (\forall\, i < n.\ E\ (xs!i))))\ True \le A\widehat{\ }n$
$\langle proof \rangle$

lemma *replicate-spmf-events*:
fixes $p :: 'a\ spmf$
fixes $n :: nat$
fixes $E :: nat \Rightarrow 'a\ option \Rightarrow bool$
assumes $\forall\, i < n.\ (spmf\ (p \gg= (\lambda x.\ return\text{-}spmf\ (E\ i\ x)))\ True = A\ i)$
shows $spmf\ (replicate\text{-}spmf\ n\ p \gg= (\lambda xs.\ return\text{-}spmf\ (\forall\, i < n.\ E\ i\ (xs!i))))$
$True = (\prod i{<}n.\ A\ i)$
$\langle proof \rangle$

lemma *replicate-spmf-same-event*:
fixes $p :: 'a\ spmf$
fixes $n :: nat$
fixes $E :: 'a\ option \Rightarrow bool$
assumes $spmf\ (p \gg= (\lambda x.\ return\text{-}spmf\ (E\ x)))\ True = A$
shows $spmf\ (replicate\text{-}spmf\ n\ p \gg= (\lambda xs.\ return\text{-}spmf\ (\forall\, i < n.\ E\ (xs!i))))\ True$
$= A\,\widehat{}\,n$
$\langle proof \rangle$

lemma *replicate-pmf-indep'*:
fixes $p :: 'a\ pmf$
fixes $n :: nat$
fixes $E :: nat \Rightarrow 'a \Rightarrow bool$
defines $rp \equiv replicate\text{-}pmf\ n\ p$
assumes $\forall\, i < n.\ pmf\ (p \gg= (\lambda x.\ return\text{-}pmf\ (E\ i\ x)))\ True = A\ i$
assumes $I \subseteq \{..{<}n\}$
assumes $\forall\, i < n.\ i \notin I \longrightarrow A\ i = 1$
shows $pmf\ (replicate\text{-}pmf\ n\ p \gg= (\lambda xs.\ return\text{-}pmf\ (\forall\, i < n.\ E\ i\ (xs!i))))\ True$
$= (\prod i{\in}I.\ A\ i)$
$\langle proof \rangle$

lemma *replicate-pmf-indep''*:
fixes $p :: 'a\ pmf$
fixes $n :: nat$
fixes $E :: 'a \Rightarrow bool$
fixes $i :: nat$
defines $rp \equiv replicate\text{-}pmf\ n\ p$
defines $A \equiv pmf\ (p \gg= (\lambda x.\ return\text{-}pmf\ (E\ x)))\ True$
assumes $i < n$
shows $pmf\ (replicate\text{-}pmf\ n\ p \gg= (\lambda xs.\ return\text{-}pmf\ (E\ (xs!i))))\ True = A$
$\langle proof \rangle$

lemma *replicate-pmf-indep*:
fixes $p :: 'a\ pmf$
fixes $n :: nat$
fixes $E :: nat \Rightarrow 'a \Rightarrow bool$
defines $rp \equiv replicate\text{-}pmf\ n\ p$
shows $prob\text{-}space.indep\text{-}events\ rp\ (\lambda i.\ \{xs \in rp.\ E\ i\ (xs!i)\})\ \{..{<}n\}$
$\langle proof \rangle$

lemma *replicate-spmf-same-event-leq*:
  fixes p :: $'a$ *spmf*
  fixes n :: *nat*
  fixes E :: $'a$ *option* $\Rightarrow$ *bool*
  assumes *spmf* $(p \gg\!= (\lambda x.\ \textit{return-spmf}\ (E\ x)))$ *True* $\leq A$
  shows *spmf* $(\textit{replicate-spmf}\ n\ p \gg\!= (\lambda xs.\ \textit{return-spmf}\ (\forall\, i < n.\ E\ (xs!i))))$ *True* $\leq A\widehat{\ }n$
  $\langle proof \rangle$

lemma *pmf-of-finite-set-event*:
  fixes S :: $'a$ *set*
  fixes p :: $'a$ *pmf*
  fixes P :: $'a \Rightarrow$ *bool*
  defines $p \equiv$ *pmf-of-set* S
  assumes $S \neq \{\}$
  assumes *finite* S
  shows *pmf* $(p \gg\!= (\lambda t.\ \textit{return-pmf}\ (P\ t)))$ *True* $= (\textit{card}\ (\{t \in S.\ P\ t\}))\ /\ \textit{card}\ S$
$\langle proof \rangle$

lemma *spmf-of-finite-set-event*:
  fixes S :: $'a$ *set*
  fixes p :: $'a$ *spmf*
  fixes P :: $'a \Rightarrow$ *bool*
  defines $p \equiv$ *spmf-of-set* S
  assumes *finite* S
  shows *spmf* $(p \gg\!= (\lambda t.\ \textit{return-spmf}\ (P\ t)))$ *True* $= (\textit{card}\ (\{t \in S.\ P\ t\}))\ /\ \textit{card}\ S$
$\langle proof \rangle$

end
theory *Hidden-Number-Problem*
  imports
    HOL−Number-Theory.Number-Theory
    Babai-Nearest-Plane.Babai-Correct
    Misc-PMF

begin

hide-fact (**open**) *Finite-Cartesian-Product.mat-def*
hide-const (**open**) *Finite-Cartesian-Product.mat*
hide-const (**open**) *Finite-Cartesian-Product.row*
hide-fact (**open**) *Finite-Cartesian-Product.row-def*
hide-const (**open**) *Determinants.det*
hide-fact (**open**) *Determinants.det-def*
hide-type (**open**) *Finite-Cartesian-Product.vec*
hide-const (**open**) *Finite-Cartesian-Product.vec*
hide-fact (**open**) *Finite-Cartesian-Product.vec-def*
hide-const (**open**) *Finite-Cartesian-Product.transpose*

hide-fact (**open**) *Finite-Cartesian-Product.transpose-def*
unbundle *no inner-syntax*
unbundle *no vec-syntax*
hide-const (**open**) *Missing-List.span*
hide-const (**open**)
dependent
independent
real-vector.representation
real-vector.subspace
span
real-vector.extend-basis
real-vector.dim
hide-const (**open**) *orthogonal*
no-notation *fps-nth* (**infixl** $ *75*)

2 General helper lemmas

lemma *uminus-sq-norm*:
fixes $u\ v$ :: *rat vec*
assumes *dim-vec* $u =$ *dim-vec* v
shows *sq-norm* $(u - v) =$ *sq-norm* $(v - u)$
⟨*proof*⟩

lemma *smult-sub-distrib-vec*:
assumes $v \in$ *carrier-vec* q $w \in$ *carrier-vec* q
shows $(a::'a::ring) \cdot_v (v - w) = a \cdot_v v - a \cdot_v w$
⟨*proof*⟩

lemma *set-list-subset*:
fixes l :: *'a list*
fixes S :: *'a set*
assumes $\forall\, i \in \{0..<length\ l\}.\ l\ !\ i \in S$
shows *set* $l \subseteq S$
⟨*proof*⟩

lemma *nat-subset-inf*:
fixes A :: *int set*
assumes $A \subseteq \mathbb{N}$
assumes $A \neq \{\}$
shows *Inf* $A \in A$
⟨*proof*⟩

lemma *cong-set-subseteq*:
fixes $a\ b\ m$ :: *'a* :: {*unique-euclidean-ring,abs*}
defines $S \equiv \{|a + z * m| \mid z.\ True\}$
defines $S' \equiv \{|b + z * m| \mid z.\ True\}$
assumes $[a = b]\ (mod\ m)$
shows $S \subseteq S'$

⟨*proof*⟩

lemma *cong-set-eq*:
  fixes $a\ b\ m :: 'a :: \{unique\text{-}euclidean\text{-}ring,abs\}$
  defines $S \equiv \{|a + z * m| \mid z.\ True\}$
  defines $S' \equiv \{|b + z * m| \mid z.\ True\}$
  assumes $[a = b]\ (mod\ m)$
  shows $S = S'$
  ⟨*proof*⟩

context *vec-module*
begin

lemma *sumlist-distrib*:
  fixes $ys :: ('a::comm\text{-}ring\text{-}1)\ vec\ list$
  assumes $\bigwedge w.\ List.member\ ys\ w \Longrightarrow dim\text{-}vec\ w = n$
  shows $k \cdot_v sumlist\ ys = sumlist\ (map\ (\lambda i.\ k \cdot_v i)\ ys)$
  ⟨*proof*⟩

lemma *smult-in-lattice-of*:
  fixes $lattice\text{-}basis :: ('a::comm\text{-}ring\text{-}1)\ vec\ list$
  assumes $\bigwedge w.\ List.member\ lattice\text{-}basis\ w \Longrightarrow dim\text{-}vec\ w = n$
  fixes $init\text{-}vec :: ('a::comm\text{-}ring\text{-}1)\ vec$
  fixes $k :: 'a::comm\text{-}ring\text{-}1$
  assumes $init\text{-}vec \in lattice\text{-}of\ lattice\text{-}basis$
  shows $k \cdot_v init\text{-}vec \in lattice\text{-}of\ ((map\ ((\cdot_v)\ k)\ lattice\text{-}basis))$
⟨*proof*⟩

end

lemma *filter-distinct-sorted*:
  fixes $l :: ('a::linorder)\ list$
  fixes $A :: 'a\ set$
  defines $P \equiv (\lambda x.\ x \in A)$
  defines $n \equiv length\ l$
  assumes *sorted l*
  assumes *distinct l*
  assumes $A \subseteq set\ l$
  shows $filter\ P\ l = sorted\text{-}list\text{-}of\text{-}set\ A$
  ⟨*proof*⟩

lemma *filter-or*:
  fixes $n\ a\ b$
  fixes $l :: nat\ list$
  defines $l \equiv [0..<n]$

defines $P \equiv (\lambda x.\ x = a \lor x = b)$
assumes $a < b$
assumes $b < \mathit{length}\ l$
shows $\mathit{filter}\ P\ l = [a,\ b]$
$\langle \mathit{proof} \rangle$

2.1 Casting lemmas

lemma *int-rat-real-casting-helper*:
assumes $a = \mathit{rat\text{-}of\text{-}int}\ b$
shows $\mathit{real\text{-}of\text{-}rat}\ a = \mathit{real\text{-}of\text{-}int}\ b$
$\langle \mathit{proof} \rangle$

lemma *casting-expansion-aux*:
shows $\mathit{int\text{-}of\text{-}rat}\ (\mathit{rat\text{-}of\text{-}int}\ w1 + \mathit{rat\text{-}of\text{-}int}\ w2) = w1 + w2$
$\langle \mathit{proof} \rangle$

lemma *casting-expansion*:
assumes $\mathit{dim\text{-}vec}\ w1 = \mathit{dim\text{-}vec}\ w2$
assumes $\exists\, w2\text{-}vec.\ w2 = \mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ w2\text{-}vec$
shows $\mathit{map\text{-}vec}\ \mathit{int\text{-}of\text{-}rat}\ ((\mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ w1) + w2) =$
$\mathit{map\text{-}vec}\ \mathit{int\text{-}of\text{-}rat}\ (\mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ w1) + (\mathit{map\text{-}vec}\ \mathit{int\text{-}of\text{-}rat}\ w2)$
$\langle \mathit{proof} \rangle$

lemma *casting-sum-aux*:
assumes $\mathit{dim\text{-}vec}\ k1 = \mathit{dim\text{-}vec}\ k2$
shows $(\mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ k1) + (\mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ k2) =$
$\mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ (k1 + k2)$
$\langle \mathit{proof} \rangle$

lemma *casting-sum-lemma*:
assumes $\bigwedge \mathit{mem}.\ \mathit{List.member}\ w\ \mathit{mem} \Longrightarrow \mathit{dim\text{-}vec}\ \mathit{mem} = n$
assumes $\bigwedge \mathit{mem}.\ \mathit{List.member}\ w\ \mathit{mem} \Longrightarrow$
$(\exists\, k{::}\mathit{int}\ \mathit{vec}.\ \mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ k = \mathit{mem})$
shows $(\exists\, k{::}\mathit{int}\ \mathit{vec}.\ (\mathit{abelian\text{-}monoid.sumlist}\ (\mathit{module\text{-}vec}\ \mathit{TYPE}(\mathit{rat})\ n)\ w) =$
$\mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ k)$
$\langle \mathit{proof} \rangle$

lemma *casting-lattice-aux*:
assumes $\exists\, k{::}\mathit{int}\ \mathit{vec}.\ \mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ k = v$
assumes $\mathit{map\text{-}vec}\ \mathit{int\text{-}of\text{-}rat}\ v =$
$\mathit{map\text{-}vec}\ \mathit{int\text{-}of\text{-}rat}\ (\mathit{abelian\text{-}monoid.sumlist}\ (\mathit{module\text{-}vec}\ \mathit{TYPE}(\mathit{rat})\ n)\ w)$
assumes $\bigwedge \mathit{mem}.\ \mathit{List.member}\ w\ \mathit{mem} \Longrightarrow$
$(\exists\, k{::}\mathit{int}\ \mathit{vec}.\ \mathit{map\text{-}vec}\ \mathit{rat\text{-}of\text{-}int}\ k = \mathit{mem})$
assumes $\bigwedge \mathit{mem}.\ \mathit{List.member}\ w\ \mathit{mem} \Longrightarrow \mathit{dim\text{-}vec}\ \mathit{mem} = n$
shows $\mathit{map\text{-}vec}\ \mathit{int\text{-}of\text{-}rat}\ v =$

abelian-monoid.sumlist (module-vec TYPE(int) n)
(map (map-vec int-of-rat) w)
⟨*proof*⟩

lemma *in-lattice-casting*:
assumes ⋀*mem. List.member qs mem* ⟹ (∃ *k::int vec. map-vec rat-of-int k* = *mem*)
assumes (⋀*mem. List.member qs mem* ⟹ *dim-vec mem* = *n*)
assumes *v* ∈ *vec-module.lattice-of n qs*
shows ∃ *k. map-vec rat-of-int k* = *v*
⟨*proof*⟩

lemma *casting-lattice-lemma-aux2*:
assumes ⋀*mem. List.member qs mem* ⟹
(∃ *k::int vec. map-vec rat-of-int k* = *mem*)
shows (*map* (*map-vec int-of-rat*) (*map* (λ*i. rat-of-int* (*c i*) $\cdot_v$ *qs* ! *i*)
[*0..<length qs*])) =
(*map* (λ*i. of-int* (*c i*) $\cdot_v$ *map* (*map-vec int-of-rat*) *qs* ! *i*) [*0..<length qs*])
⟨*proof*⟩

lemma *casting-lattice-lemma*:
fixes *v* :: *rat vec*
fixes *qs* :: *rat vec list*
assumes ∃ *k::int vec. map-vec rat-of-int k* = *v*
assumes ⋀*mem. List.member qs mem* ⟹
(∃ *k::int vec. map-vec rat-of-int k* = *mem*)
assumes *dim-vecs*: ⋀*mem. List.member qs mem* ⟹ *dim-vec mem* = *n*
assumes *v* ∈ *vec-module.lattice-of n qs*
shows *map-vec int-of-rat v*
∈ *vec-module.lattice-of n* (*map* (*map-vec int-of-rat*) *qs*)
⟨*proof*⟩

3 HNP adversary locale

locale *hnp-adversary* =
fixes *d p* :: *nat*
begin

type-synonym *adversary* = (*nat* × *nat*) *list* ⇒ *nat*

definition *int-gen-basis* :: *nat list* ⇒ *int vec list* **where**
int-gen-basis ts = *map* (λ*i.* (*p^2* $\cdot_v$ (*unit-vec* (*d+1*) *i*))) [*0..<d*]
@ [*vec-of-list* ((*map* (λ*x.* (*of-nat p*) ∗ *x*) *ts*) @ [*1*])]

fun *int-to-nat-residue* :: *int* ⇒ *nat* ⇒ *nat*
where *int-to-nat-residue I modulus* = *nat* (*I mod modulus*)

definition *ts-from-pairs* :: (*nat* × *nat*) *list* ⇒ *nat list* **where**

ts-from-pairs pairs = *map fst pairs*

definition *scaled-uvec-from-pairs* :: (*nat* × *nat*) *list* ⇒ *int vec* **where**
scaled-uvec-from-pairs pairs = *vec-of-list* ((*map* (λ(*a*,*b*). *p*∗*b*) *pairs*) @ [*0*])

fun $\mathcal{A}$-*vec* :: (*nat* × *nat*) *list* ⇒ *int vec* **where**
$\mathcal{A}$-*vec pairs* = (*full-babai*
(*int-gen-basis* (*ts-from-pairs pairs*))
(*map-vec rat-of-int* (*scaled-uvec-from-pairs pairs*))
(*4*/*3*))

fun $\mathcal{A}$ :: *adversary* **where**
$\mathcal{A}$ *pairs* = *int-to-nat-residue* (($\mathcal{A}$-*vec pairs*)$*d*) *p*

end

4 HNP locales

4.0.1 Arithmetic locale

locale *hnp-arith* =
fixes *n* α *d* *p* *k* :: *nat*
assumes *p*: *prime p*
assumes α: α ∈ {*1*..<*p*}
assumes *d*: *d* = *2* ∗ *ceiling* (*sqrt n*)
assumes *n*: *n* = *ceiling* (*log 2 p*)
assumes *k*: *k* = *ceiling* (*sqrt n*) + *ceiling* (*log 2 n*)
assumes *n-big*: *961* < *n*
begin

definition μ :: *nat* **where**
μ ≡ *nat* (*ceiling* (*1*/*2* ∗ (*sqrt n*)) + *3*)

lemma μ: μ = (*ceiling* (*1*/*2* ∗ (*sqrt n*)) + *3*)
⟨*proof*⟩

lemma *int-p-prime*: *prime* (*int p*) ⟨*proof*⟩

lemma *p-geq-2*: *p* ≥ *2* ⟨*proof*⟩

lemma *n-geq-1*: *n* ≥ *1*
⟨*proof*⟩

lemma μ-*le-k*:
shows μ + *1* ≤ *k*
⟨*proof*⟩

lemma *k-plus-1-lt*:
shows *k* + *1* < *log 2 p*

$\langle proof \rangle$

lemma *p-k-le-p-mu*:
  shows $p/(2\hat{}k) \le p/(2\hat{}(\mu + 1))$
$\langle proof \rangle$

lemma *k-geq-1*: $k \ge 1$ $\langle proof \rangle$

lemma *final-ineq*:
  $(((4::real)/3)$ *powr* $((d + 1)/2) * (11/10) * (d + 1) * (p$ / $(2$ *powr* $($*real* $k -$
$1)))$
   $< p$ / $(2$ *powr* $\mu))$
$\langle proof \rangle$

end

4.1 Main HNP locale

locale *hnp* = *hnp-arith* n α d p k + *hnp-adversary* d p **for** n α d p k +
  fixes *msb-p* :: *nat* $\Rightarrow$ *nat*
  assumes *msb-p-dist*: $\bigwedge x.$ $|$*int* x $-$ *int* $($*msb-p* $x)| < p$ / $(2\hat{}k)$
begin

4.2 Uniqueness lemma

sublocale *vec-space* *TYPE*(*rat*) $d + 1$ $\langle proof \rangle$

lemma *sumlist-index-commute*:
  fixes *Lst* :: *rat vec list*
  fixes i :: *nat*
  assumes *set Lst* $\subseteq$ *carrier-vec* $(d + 1)$
  assumes $i < (d + 1)$
  shows (*sumlist Lst*)\$$i$ = *sum-list* (*map* ($\lambda j.$ (*Lst*!j)\$$i$) [$0..<$(*length Lst*)])
  $\langle proof \rangle$

definition *ts-pmf* **where** *ts-pmf* = *replicate-pmf* d (*pmf-of-set* $\{1..<p\}$)

lemma *set-pmf-ts*: *set-pmf ts-pmf* = $\{l.$ *length* $l = d \wedge (\forall i < d.\ l!i \in \{1..<p\})\}$
$\langle proof \rangle$

definition *ts-to-as* :: *nat list* $\Rightarrow$ *nat list* **where**
  ts-to-as ts = (*map* (*msb-p* $\circ$ ($\lambda t.$ ($\alpha * t$) *mod* p)) *ts*)

definition *ts-to-u* :: *nat list* $\Rightarrow$ *rat vec* **where**
  ts-to-u ts = *vec-of-list* (*map of-nat* (*ts-to-as ts*) @ [0])

lemma *ts-to-u-alt*:
  ts-to-u ts = *vec-of-list* ((*map* (*of-nat* $\circ$ *msb-p* $\circ$ ($\lambda t.$ ($\alpha * t$) *mod* p)) *ts*) @ [0])
  $\langle proof \rangle$

lemma *u-carrier*: *length ts* = *d* $\Longrightarrow$ *ts-to-u ts* $\in$ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *ts-to-u-carrier*:
fixes *ts* :: *nat list*
shows (*ts-to-u ts*) $\in$ *carrier-vec* ((*length ts*) + *1*)
⟨*proof*⟩

4.2.1 Lattice construction and lemmas

definition *p-vecs* :: *rat vec list* **where**
p-vecs = *map* (λ*i*. *of-int-hom.vec-hom* ((*of-nat p*) $\cdot_v$ (*unit-vec* (*d*+*1*) *i*))) [*0*..<*d*]

lemma *length-p-vecs*: *length p-vecs* = *d* ⟨*proof*⟩

lemma *p-vecs-carrier*: $\forall$ *v* $\in$ *set p-vecs*. *dim-vec v* = *d* + *1* ⟨*proof*⟩

lemma *lincomb-of-p-vecs-last*: (*lincomb-list* (*of-int* $\circ$ *cs*) *p-vecs*)$*d* = *0* (**is** *?lhs* = *0*)
⟨*proof*⟩

definition *gen-basis* :: *nat list* $\Rightarrow$ *rat vec list* **where**
gen-basis ts = *p-vecs* @ [*vec-of-list* ((*map of-nat ts*) @ [*1* / (*of-nat p*)])]

lemma *gen-basis-length*: *length* (*gen-basis ts*) = *d* + *1* ⟨*proof*⟩

lemma *gen-basis-units*:
assumes *i* < *d*
assumes *j* < *d* + *1*
shows ((*gen-basis ts*)!*i*)$*j* = *of-nat* (*if i* = *j then p else 0*) (**is** (*?x*)$*j* = -)
⟨*proof*⟩

definition *int-gen-lattice* :: *nat list* $\Rightarrow$ *int vec set* **where**
int-gen-lattice ts = *vec-module.lattice-of* (*d* + *1*) (*int-gen-basis ts*)

definition *gen-lattice* :: *nat list* $\Rightarrow$ *rat vec set* **where**
gen-lattice ts = *vec-module.lattice-of* (*d* + *1*) (*gen-basis ts*)

definition *close-vec* :: *nat list* $\Rightarrow$ *rat vec* $\Rightarrow$ *bool* **where**
close-vec ts v $\longleftrightarrow$ (*sq-norm* ((*ts-to-u ts*) − *v*) < ((*of-nat p*) / *2*^μ)^*2*)

definition *good-vec* :: *rat vec* $\Rightarrow$ *bool* **where**
good-vec v $\longleftrightarrow$ *dim-vec v* = *d* + *1* $\wedge$ ($\exists\beta$::*int*. [α = β] (*mod p*) $\wedge$ *of-rat* (*v*$*d*) = β/*p*)

definition *good-lattice* :: *nat list* $\Rightarrow$ *bool* **where**
good-lattice ts $\longleftrightarrow$ ($\forall$ *v* $\in$ *gen-lattice ts*. *close-vec ts v* $\longrightarrow$ *good-vec v*)

definition *bad-lattice* :: *nat list* $\Rightarrow$ *bool* **where**

$bad\text{-}lattice\ ts \longleftrightarrow \neg\ good\text{-}lattice\ ts$

definition *sampled-lattice-good* :: *bool pmf* **where**
sampled-lattice-good = *do* {
ts ← *ts-pmf*;
return-pmf (*good-lattice ts*)
}

interpretation *vec-int*: *vec-module TYPE*(*int*) *d* + *1* ⟨*proof*⟩

lemma *int-gen-basis-carrier*:
fixes *ts* :: *nat list*
assumes *length ts* = *d*
shows *set* (*int-gen-basis ts*) ⊆ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *int-gen-lattice-carrier*:
fixes *ts* :: *nat list*
assumes *length ts* = *d*
shows *int-gen-lattice ts* ⊆ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *gen-basis-vecs-carrier*:
fixes *ts* :: *nat list*
fixes *i* :: *nat*
assumes *length ts* = *d*
assumes *i* ∈ {*0*..< *d* + *1*}
shows (*gen-basis ts*) ! *i* ∈ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *gen-basis-carrier*:
fixes *ts* :: *nat list*
assumes *length ts* = *d*
shows *set* (*gen-basis ts*) ⊆ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *gen-lattice-carrier*:
fixes *ts* :: *nat list*
assumes *length ts* = *d*
shows *gen-lattice ts* ⊆ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *sampled-lattice-good-map-pmf*: *sampled-lattice-good* = *map-pmf good-lattice ts-pmf*
⟨*proof*⟩

lemma *coordinates-of-gen-lattice*:
fixes *ts* :: *nat list*

fixes *c* :: *nat* ⇒ *int*
fixes *i* :: *nat*
assumes $i \leq$ *length ts*
assumes *length ts* = *d*
shows (*sumlist* (*map* (λ*i*. *of-int* (*c i*) $\cdot_v$ ((*gen-basis ts*) ! *i*)) [*0* ..< *length* (*gen-basis ts*)]))$*i*
= (*if* (*i* = *d*) *then* (*rat-of-int* (*c d*) / *rat-of-nat p*) *else* *rat-of-int* ((*c d*) ∗ *ts*!*i* + (*c i*)∗*p*))
⟨*proof*⟩

lemma *gen-lattice-int-gen-lattice-vec*:
fixes *scaled-v* :: *int vec*
fixes *ts* :: *nat list*
assumes *length ts* = *d*
assumes (*scaled-v* ∈ *int-gen-lattice ts*)
shows ((*1*/(*of-nat p*)) $\cdot_v$ (*map-vec rat-of-int scaled-v*) ∈ (*gen-lattice ts*))
⟨*proof*⟩

definition *t-vec* :: *nat list* ⇒ *rat vec* **where**
t-vec ts = *vec-of-list* ((*map of-nat ts*) @ [*1* / (*of-nat p*)])

lemma *t-vec-dim*: *dim-vec* (*t-vec ts*) = *length ts* + *1*
⟨*proof*⟩

lemma *t-vec-last*: *length ts* = *d* ⟹ (*t-vec ts*)$*d* = *1* / (*of-nat p*)
⟨*proof*⟩

definition *z-vecs* :: *int vec set* **where**
z-vecs = {*v*. *dim-vec v* = *d* + *1* ∧ *v*$*d* = *0*}

definition *vec-class* :: *nat list* ⇒ *int* ⇒ *rat vec set* **where**
vec-class ts β = {(*of-int* β) $\cdot_v$ (*t-vec ts*) + *lincomb-list* (*of-int* ∘ *cs*) *p-vecs* | *cs* :: *nat* ⇒ *int*. *True*}

definition *vec-class-mod-p* :: *nat list* ⇒ *int* ⇒ *rat vec set* **where**
vec-class-mod-p ts β = ⋃{*vec-class ts* β′ | β′. [β = β′] (*mod p*)}

lemma *vec-class-carrier*:
assumes *length ts* = *d*
shows *vec-class ts* β ⊆ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *vec-class-mod-p-carrier*:
assumes *length ts* = *d*
shows *vec-class-mod-p ts* β ⊆ *carrier-vec* (*d* + *1*)
⟨*proof*⟩

lemma *vec-class-last*:
assumes *length ts* = *d*

assumes $v \in$ *vec-class ts* β
shows $v\$d$ = *rat-of-int* β / *rat-of-int p*
⟨*proof*⟩

lemma *gen-lattice-int-gen-lattice-vec′*:
fixes v :: *rat vec*
fixes *ts* :: *nat list*
assumes *length ts* = d
assumes $v \in$ *gen-lattice ts*
shows *map-vec int-of-rat* ((*of-nat p*) $\cdot_v$ v) $\in$ *int-gen-lattice ts*
map-vec rat-of-int (*map-vec int-of-rat* ((*of-nat p*) $\cdot_v$ v)) = (*rat-of-nat p*) $\cdot_v$ v
⟨*proof*⟩

lemma *gen-lattice-int-gen-lattice-closest*:
fixes *scaled-v* :: *int vec*
fixes u :: *rat vec*
fixes *ts* :: *nat list*
assumes *length ts* = d
assumes *dim-vec u* = $d + 1$
shows *real*(*p*^*2*) $*$ *Inf*{*real-of-rat* (*sq-norm* ($x - u$))| x. $x \in$ *gen-lattice ts*}
= *babai.closest-distance-sq* (*int-gen-basis ts*) ((*rat-of-nat p*) $\cdot_v$ u)
⟨*proof*⟩

lemma *close-vector-exists*:
fixes *ts* :: *nat list*
assumes *length ts* = d
shows $\exists\, w \in$(*gen-lattice ts*). *sq-norm* ((*ts-to-u ts*) $-$ w) $\leq$ (*of-nat* ((d+*1*) $*$ *p*^*2*))/*2*^(*2*$*$*k*)
⟨*proof*⟩

lemma *gen-lattice-dim*:
assumes *ts* $\in$ *set-pmf ts-pmf*
assumes $v \in$ *gen-lattice ts*
shows *dim-vec v* = $d + 1$
⟨*proof*⟩

lemma *vec-class-union*:
fixes *ts* :: *nat list*
assumes *ts* $\in$ *set-pmf ts-pmf*
defines $L \equiv$ *gen-lattice ts*
shows $L = \bigcup$ {*vec-class ts* β | β. *True*}
⟨*proof*⟩

lemma *vec-class-mod-p-union*:
fixes *ts* :: *nat list*
assumes *ts* $\in$ *set-pmf ts-pmf*
defines $L \equiv$ *gen-lattice ts*
shows $L = \bigcup$ {*vec-class-mod-p ts* β | β. $\beta \in$ {*0*..<*p*::*int*}} (**is** - = *?rhs*)

⟨*proof*⟩

4.2.2 dist-p definition and lemmas

definition *dist-p* :: *int* ⇒ *int* ⇒ *int* **where**
  dist-p i j = *Inf* {*abs* (*i* − *j* + *z* ∗ (*of-nat p*)) | *z*. *True*}

lemma *dist-p-well-defined*:
  fixes *i* :: *int*
  fixes *j* :: *int*
  shows *dist-p i j* ∈ {*abs* (*i* − *j* + *z* ∗ (*of-nat p*)) | *z*. *True*} (**is** - ∈ *?S*)
⟨*proof*⟩

lemma *dist-p-set-bdd-below*:
  fixes *i j* :: *int*
  shows ∀ *x* ∈ {*abs* (*i* − *j* + *z* ∗ (*of-nat p*)) | *z*. *True*}. *0* ≤ *x*
  ⟨*proof*⟩

lemma *dist-p-le*:
  fixes *i j* :: *int*
  shows ∀ *x* ∈ {*abs* (*i* − *j* + *z* ∗ (*of-nat p*)) | *z*. *True*}. *dist-p i j* ≤ *x* (**is** ∀ *x* ∈ *?S*. -)
⟨*proof*⟩

lemma *dist-p-equiv*:
  fixes *i* :: *int*
  fixes *j* :: *int*
  shows [*dist-p i j* = *i* − *j*] (*mod p*) ∨ [*dist-p i j* = *j* − *i*] (*mod p*)
⟨*proof*⟩

lemma *dist-p-equiv′*:
  fixes *i* :: *int*
  fixes *j* :: *int*
  shows *dist-p i j* = *min* ((*i* − *j*) *mod p*) ((*j* − *i*) *mod p*)
⟨*proof*⟩

lemma *dist-p-equiv′′*:
  fixes *i* :: *int*
  fixes *j* :: *int*
  shows *dist-p i j* = ((*i* − *j*) *mod p*) ∨ *dist-p i j* = ((*j* − *i*) *mod p*)
  ⟨*proof*⟩

lemma *dist-p-instances-help*:
  fixes *i* :: *int*
  fixes *j* :: *int*
  fixes *dist* :: *int*
  assumes *coprime* (*i* − *j*) *p*
  shows *card* {*t* ∈ {*1*..<*p*}. (*dist-p* (*t*∗*i*) (*t*∗*j*)) = *dist*} ≤ *2*
⟨*proof*⟩

lemma *dist-p-nat*:
fixes i :: *int*
fixes j :: *int*
shows *dist-p* i $j \in \mathbb{N}$
⟨*proof*⟩

lemma *dist-p-nat′*:
shows *nat* (*dist-p* i j) = *dist-p* i j
⟨*proof*⟩

lemma *dist-p-instances*:
fixes i :: *int*
fixes j :: *int*
fixes *bound* :: *rat*
assumes $i \neq j$
assumes *coprime* $(i - j)$ p
assumes $0 <$ *bound*
shows *rat-of-nat* (*card* $\{t \in \{1..<p\}$. *rat-of-int* (*dist-p* $(t*i)$ $(t*j)$) $\leq$ *bound*$\}$) $\leq$ $2*$*bound*
⟨*proof*⟩

lemma *dist-p-smallest*:
fixes β *ts* i
assumes *ts* $\in$ *set-pmf ts-pmf*
assumes $v \in$ *vec-class-mod-p ts* β
assumes $i < d$
defines $u \equiv$ *ts-to-u ts*
shows (*of-int* (*dist-p* (*int* (*ts* ! i) $*$ β) (*int* (*ts-to-as ts* ! i))))$^2 \leq ((u - v)\$i)^2$
(**is** (*of-int* ?d)$^2 \leq$ -)
⟨*proof*⟩

lemma *dist-p-diff-helper*:
fixes a b b'
assumes $b \geq b'$
shows $|$*dist-p* a b $-$ *dist-p* a $b'| \leq b - b'$
⟨*proof*⟩

lemma *dist-p-diff*:
fixes a b b'
shows $|$*dist-p* a b $-$ *dist-p* a $b'| \leq |b - b'|$
⟨*proof*⟩

4.2.3 Uniquness lemma argument

definition *good-beta* **where**
good-beta ts $\beta \longleftrightarrow$ ($\forall$ $v \in$ *vec-class-mod-p ts* β. *close-vec ts* $v \longrightarrow$ *good-vec* v)

definition *bad-beta* **where**

bad-beta ts $\beta \longleftrightarrow \neg$ *good-beta ts* β

definition *some-bad-beta* **where**
some-bad-beta ts $\longleftrightarrow$ ($\exists\, \beta \in \{0..<p::int\}$. *bad-beta ts* β)

definition *bad-beta-union* **where**
bad-beta-union = ($\bigcup \beta \in \{0..<p::int\}$. $\{ts.$ *bad-beta ts* $\beta\}$)

lemma *reduction-1*:
assumes *ts* $\in$ *set-pmf ts-pmf*
assumes $\neg$ *good-lattice ts*
shows *some-bad-beta ts*
⟨*proof*⟩

lemma *reduction-1-pmf*:
pmf sampled-lattice-good False $\leq$ *pmf* (*ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* (*some-bad-beta ts*))) *True*
⟨*proof*⟩

lemma *reduction-2*: $\{ts.$ *some-bad-beta ts*$\}$ $\subseteq$ *bad-beta-union*
⟨*proof*⟩

lemma *reduction-2-pmf*:
pmf (*ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* (*ts* $\in$ $\{ts.$ *some-bad-beta ts*$\}$))) *True*
$\leq$ *pmf* (*ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* (*ts* $\in$ *bad-beta-union*))) *True*
⟨*proof*⟩

lemma *bad-beta-union-bound*:
pmf (*ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* (*ts* $\in$ *bad-beta-union*))) *True*
$\leq$ ($\sum \beta \in \{0..<p::int\}$. *pmf* (*ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* (*ts* $\in$ $\{ts.$ *bad-beta ts* $\beta\}$))) *True*)
(**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

lemma *fixed-beta-close-vec-dist-p*:
fixes β *ts*
assumes *ts* $\in$ *set-pmf ts-pmf*
assumes *v* $\in$ *vec-class-mod-p ts* β
assumes *close-vec ts v*
defines *u* $\equiv$ *ts-to-u ts*
shows $\forall\, i < d$. *real-of-int* (*dist-p* (*int* (*ts* ! *i*) $*$ β) (*int* (*ts-to-as ts* ! *i*))) $<$ *real p* / 2 ^ μ
⟨*proof*⟩

lemma *fixed-beta-bad-prob-arith-helper*:
defines *T-lwr-bnd* $\equiv$ (*rat-of-nat* ($p - 1$) $-$ 2 $*$ (2 $*$ (*rat-of-nat p*))/(2^μ))
shows $1 - 5/(2\hat{}\mu) \leq$ *real-of-rat T-lwr-bnd* / ($p - 1$)
⟨*proof*⟩

lemma *dist-p-helper*:
  fixes *ts i*
  assumes *ts*: $ts \in$ *set-pmf ts-pmf*
  assumes *i*: $i < d$
  assumes *dist*: *dist-p* (*int* (*ts*!*i*) $*$ β) ((*ts-to-as ts*)!*i*) $<$ $p/(2\hat{}\mu)$
  shows *dist-p* ((*ts*!*i*) $*$ β) ((*ts*!*i*) $*$ α) $\leq$ $(2{*}p)/(2\hat{}\mu)$
⟨*proof*⟩

lemma *prob-A-helper*:
  fixes β :: *int*
  defines [*simp*]: *t-pmf* $\equiv$ *pmf-of-set* $\{1..{<}p\}$
  defines [*simp*]: *A-cond* $\equiv$ $\lambda t.$ $(2{*}p)/(2\hat{}\mu)$ $<$ *dist-p* (β $*$ t) (α $*$ t)
  defines [*simp*]: *A-pmf* $\equiv$ *t-pmf* $\gg=$ ($\lambda t.$ *return-pmf* (*A-cond t*))
  defines [*simp*]: $A \equiv$ *pmf A-pmf True*
  assumes β: $\beta \in \{0..{<}p{::}int\}$
  assumes *False*: $\neg$ ($\beta = \alpha$ *mod p*)
  shows $(1 - A) \leq (5\ /\ (2\hat{}\mu))$
⟨*proof*⟩

lemma *prob-A-helper'*:
  fixes β :: *int*
  defines [*simp*]: *t-pmf* $\equiv$ *pmf-of-set* $\{1..{<}p\}$
  defines [*simp*]: *A-cond* $\equiv$ $\lambda t.$ $(2{*}p)/(2\hat{}\mu)$ $<$ *dist-p* (β $*$ t) (α $*$ t)
  defines [*simp*]: *A-pmf* $\equiv$ *t-pmf* $\gg=$ ($\lambda t.$ *return-pmf* (*A-cond t*))
  defines [*simp*]: $A \equiv$ *pmf A-pmf True*
  assumes β: $\beta \in \{0..{<}p{::}int\}$
  assumes *False*: $\neg$ ($\beta = \alpha$ *mod p*)
  shows $(1 - A)\hat{}d \leq (5\ /\ (2\hat{}\mu))\hat{}d$
  ⟨*proof*⟩

lemma *fixed-beta-bad-prob*:
  fixes β
  assumes $\beta \in \{0..{<}p{::}int\}$
  defines $M \equiv$ *ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* ($ts \in \{ts.$ *bad-beta ts* $\beta\}$))
  shows $\beta = \alpha$ *mod p* $\Longrightarrow$ *pmf M True* $= 0$
        $\neg$ ($\beta = \alpha$ *mod p*) $\Longrightarrow$ *pmf M True* $\leq (5/(2\hat{}\mu))\hat{}d$
⟨*proof*⟩

lemma *bad-beta-union-bound-pmf*:
  pmf (*ts-pmf* $\gg=$ ($\lambda ts.$ *return-pmf* ($ts \in$ *bad-beta-union*))) *True* $\leq (p - 1)\ *$
$(5/(2\hat{}\mu))\hat{}d$
⟨*proof*⟩

lemma *sampled-lattice-unlikely-bad*:
  shows *pmf sampled-lattice-good False* $\leq (p - 1) * ((5/(2\hat{}\mu))\hat{}d)$
  ⟨*proof*⟩

4.2.4 Main uniqueness lemma statement

lemma *sampled-lattice-likely-good*: *pmf sampled-lattice-good True* $\geq$ *1/2*
⟨*proof*⟩

4.3 Main theorem

4.3.1 Oracle definition

definition $\mathcal{O}$ :: *nat* $\Rightarrow$ *nat* **where**
$\mathcal{O}$ *t* = *msb-p* ((*α* ∗ *t*) *mod p*)

4.3.2 Apply Babai lemmas to adversary

We use Babai lemmas to show that the adversary wins when the *ts* generate a good lattice.

lemma *gen-basis-assms*:
fixes *ts* :: *nat list*
assumes *length ts* = *d*
shows *LLL.LLL-with-assms* (*d*+*1*) (*d*+*1*) (*int-gen-basis ts*) (*4/3*)
⟨*proof*⟩

definition *u-vec-is-msbs* :: (*nat*×*nat*) *list* $\Rightarrow$ *bool* **where**
u-vec-is-msbs pairs = ((*of-nat p*) $\cdot_v$ *ts-to-u* (*ts-from-pairs pairs*) = *of-int-hom.vec-hom* (*scaled-uvec-from-pairs pairs*))

lemma *updated-full-Babai-correct-int-target*:
assumes *full-babai-with-assms* (*map-vec rat-of-int target*) (*d* + *1*) *fs-init* (*4/3*)
shows *real-of-int* (*sq-norm* ((*full-babai fs-init* (*map-vec rat-of-int target*) (*4/3*)) − *target*))
≤ ((*4/3*)^(*d* + *1*) ∗ (*d* + *1*)) ∗ *11/10* ∗ *babai.closest-distance-sq fs-init* (*map-vec rat-of-int target*) ∧
(*full-babai fs-init* (*map-vec rat-of-int target*) (*4/3*)) ∈ *vec-module.lattice-of* (*d* + *1*) *fs-init*
⟨*proof*⟩

lemma *ad-output-vec-close*:
fixes *pairs* :: (*nat*×*nat*) *list*
assumes *length pairs* = *d*
assumes *u-vec-is-msbs pairs*
shows *real-of-int*(*sq-norm*
(($\mathcal{A}$-*vec pairs*)
− (*scaled-uvec-from-pairs pairs*))) ≤
(*4* / *3*) ^ (*d* + *1*) ∗ *real* (*d* + *1*) ∗ *11* / *10* ∗ *p*^*2* ∗ (*of-nat* ((*d*+*1*) ∗ *p*^*2*))/*2*^(*2*∗*k*)
∧ ($\mathcal{A}$-*vec pairs*) ∈ *int-gen-lattice* (*ts-from-pairs pairs*)
⟨*proof*⟩

lemma *ad-output-vec-class*:
fixes *pairs* :: (*nat* × *nat*) *list*

assumes *length pairs* = *d*
assumes *u-vec-is-msbs pairs*
shows *sq-norm* (($1/($*rat-of-nat p*$)$)$\cdot_v$(*map-vec rat-of-int* ($\mathcal{A}$-*vec pairs*)) − (*ts-to-u* (*ts-from-pairs pairs*))) < ((*of-nat p*)/2^(μ))^2
∧ ((1/(*rat-of-nat p*))$\cdot_v$(*map-vec rat-of-int* ($\mathcal{A}$-*vec pairs*)) ∈ *vec-class-mod-p* (*ts-from-pairs pairs*) ($\mathcal{A}$ *pairs*))
⟨*proof*⟩

If we get a good lattice (which is likely), the adversary finds the hidden number

lemma *hnp-adversary-exists-helper*:
assumes *ts* ∈ *set-pmf ts-pmf*
defines *ts-Ots* ≡ *map* (λ*t*. (*t*, $\mathcal{O}$ *t*)) *ts*
assumes *good-lattice ts*
shows α = $\mathcal{A}$ *ts-Ots*
⟨*proof*⟩

4.3.3 Main theorem statement

The cryptographic game which the adversary should win with high probability.

definition *game* :: *adversary* ⇒ *bool pmf* **where**
game $\mathcal{A}'$ = *do* {
ts ← *replicate-pmf d* (*pmf-of-set* {1..<*p*});
return-pmf (α = $\mathcal{A}'$ (*map* (λ*t*. (*t*, $\mathcal{O}$ *t*)) *ts*))
}

The adversary finds the hidden number with probability at least 1/2.

theorem *hnp-adversary-exists*: *pmf* (*game* $\mathcal{A}$) *True* ≥ 1/2
⟨*proof*⟩

Alternative definition of *game*, written as a *map-pmf*

definition *game'* :: *adversary* ⇒ *bool pmf* **where**
game' $\mathcal{A}'$ = *map-pmf* ((λ*ts*. (α = $\mathcal{A}'$ (*map* (λ*t*. (*t*, $\mathcal{O}$ *t*)) *ts*)))) (*replicate-pmf d* (*pmf-of-set* {1..<*p*}))

lemma *game'* = *game*
⟨*proof*⟩

end

5 Some MSB instantiations and lemmas

5.1 Bit-shift MSB

definition *msb* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
msb k n x ≡ (*x div* 2^(*n* − *k*)) ∗ 2^(*n* − *k*)

lemma *msb-dist*:
  fixes $k\ n$ :: *nat*
  assumes $k < n$
  assumes $1 \leq k$
  shows $|int\ (x) - int\ (msb\ k\ n\ x)| < 2\hat{}n\ /\ (2\hat{}k)$
⟨*proof*⟩

lemma (**in** *hnp-arith*) *msb-kp1-dist-hnp*:
  defines $msb\text{-}k \equiv msb\ (k + 1)\ n$
  shows $|int\ (x) - int\ (msb\text{-}k\ x)| < p\ /\ (2\hat{}k)$
⟨*proof*⟩

lemma *msb-kp1-valid-hnp*:
  assumes $hnp\text{-}arith\ n\ \alpha\ d\ p\ k$
  shows $hnp\ n\ \alpha\ d\ p\ k\ (msb\ (k + 1)\ n)$
  ⟨*proof*⟩

5.2 MSB-p

definition *msb-p* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$ **where**
  $msb\text{-}p\ p\ k\ x =$
    $(let\ t = (THE\ t.\ (t - 1) * (p\ /\ 2\hat{}k) \leq x \wedge x < t * p\ /\ 2\hat{}k)$
    $in\ nat\ (floor\ (t * p\ /\ 2\hat{}k)) - 1)$

lemma *msb-p-defined*:
  fixes $p\ k$ :: *nat*
  fixes x :: *nat*
  assumes $p > 0$
  assumes $k > 0$
  shows $(\exists!t.\ (t - 1) * (p\ /\ 2\hat{}k) \leq x \wedge x < t * p\ /\ 2\hat{}k)$
⟨*proof*⟩

lemma (**in** *hnp-arith*) *msb-p-dist-hnp*:
  defines $msb\text{-}k \equiv msb\text{-}p\ p\ k$
  shows $|int\ x - int\ (msb\text{-}k\ x)| < p\ /\ 2\hat{}k$
⟨*proof*⟩

lemma *msb-p-valid-hnp*:
  assumes $hnp\text{-}arith\ n\ \alpha\ d\ p\ k$
  defines $msb\text{-}k \equiv msb\text{-}p\ p\ k$
  shows $hnp\ n\ \alpha\ d\ p\ k\ msb\text{-}k$
  ⟨*proof*⟩

end
theory *Ad-Codegen-Example*
  imports *Hidden-Number-Problem*

begin

6 This theory demonstrates an example of the executable adversary.

full-babai does not need a global interpretation to be executed.

value *full-babai* [*vec-of-list* [*0*, *1*], *vec-of-list* [*2*, *3*]] (*vec-of-list* [*2.3*, *6.4*]) (*4/3*)

Let's define our d, n, p, α, and k.

abbreviation $d \equiv 72$
abbreviation $n \equiv 1279$
abbreviation $p \equiv (2{::}nat)\hat{}1279 - 1$
abbreviation $\alpha \equiv p\ div\ 3$
abbreviation $k \equiv 47$

value p
value α

Since our adversary definition is inside a locale, we need a global interpretation.

global-interpretation *ad-interp*: *hnp-adversary d p*
  defines $\mathcal{A}$ = *ad-interp*.$\mathcal{A}$
  and *int-gen-basis* = *ad-interp.int-gen-basis*
  and *int-to-nat-residue* = *ad-interp.int-to-nat-residue*
  and *ts-from-pairs* = *ad-interp.ts-from-pairs*
  and *scaled-uvec-from-pairs* = *ad-interp.scaled-uvec-from-pairs*
  and $\mathcal{A}$-*vec* = *ad-interp*.$\mathcal{A}$-*vec*
  ⟨*proof*⟩

For this example, we use our executable msb function.

abbreviation $\mathcal{O} \equiv \lambda t.\ msb\ k\ n\ ((\alpha * t)\ mod\ p)$

definition *inc-amt* :: *nat* **where** *inc-amt* = *p div 5*

fun *gen-ts* :: *nat* ⇒ *nat* ⇒ *nat list* **where**
  gen-ts 0 t = []
| *gen-ts* (*Suc i*) *t* = *t* # (*gen-ts i* ((*t* + *inc-amt*) *mod p*))

definition *gen-pairs* :: (*nat* × *nat*) *list* **where**
  gen-pairs = *map* ($\lambda t.$ (t, $\mathcal{O}$ t)) (*gen-ts d 1*)

The *gen-pairs* function generates the data that the adversary receives. We prove that the adversary is likely to be successful when the *ts* which define this data are uniformly distributed. Here, we use the *gen-ts* function to generate an explicit list of *ts*.

value *length gen-pairs*

value *gen-pairs*

value *ad-interp*.$\mathcal{A}$ *gen-pairs*

value α

end

References

[1] L. Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Comb.*, 6(1):1–13, 1986.

[2] D. Boneh and R. Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 129–142. Springer, 1996.

[3] R. Bottesch, M. W. Haslbeck, and R. Thiemann. A verified efficient implementation of the LLL basis reduction algorithm. In G. Barthe, G. Sutcliffe, and M. Veanes, editors, *LPAR*, volume 57 of *EPiC Series in Computing*, pages 164–180. EasyChair, 2018.

[4] J. Divasón, S. J. C. Joosten, R. Thiemann, and A. Yamada. A Verified Factorization Algorithm for Integer Polynomials with Polynomial Complexity. *Archive of Formal Proofs*, February 2018. https://isa-afp.org/entries/LLL_Factorization.html, Formal proof development.

[5] A. Lenstra, H. Lenstra, and L. László. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261, 12 1982.

[6] R. Thiemann, R. Bottesch, J. Divasón, M. W. Haslbeck, S. J. C. Joosten, and A. Yamada. Formalizing the LLL basis reduction algorithm and the LLL factorization algorithm in Isabelle/HOL. *J. Autom. Reason.*, 64(5):827–856, 2020.