Introduction to symmetric crypto
D. J. Bernstein

How HTTPS protects connection:

• Public-key encryption system encrypts \textit{one} secret message: a random 256-bit session key.
• Public-key signature system stops NSAI\textsc{TM} attacks.
• Fast \textit{authenticated cipher} uses the 256-bit session key to protect further messages.

Some cipher history

Introduction to symmetric crypto
D. J. Bernstein

How HTTPS protects connection:

• Public-key encryption system encrypts one secret message: a random 256-bit session key.

• Public-key signature system stops NSA/ITM attacks.

• Fast authenticated cipher uses the 256-bit session key to protect further messages.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.
Introduction to symmetric crypto
D. J. Bernstein

How HTTPS protects connection:
• Public-key encryption system encrypts one secret message: a random 256-bit session key.
• Public-key signature system stops NSAITM attacks.
• Fast authenticated cipher uses the 256-bit session key to protect further messages.

Some cipher history
1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.
1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”
Introduction to symmetric crypto

D. J. Bernstein

How HTTPS protects connection:

• Public-key encryption system encrypts one secret message: a random 256-bit session key.

• Public-key signature system stops NSAITM attacks.

• Fast authenticated cipher uses the 256-bit session key to protect further messages.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
Introduction to symmetric crypto

D. J. Bernstein

How HTTPS protects connection:

• Public-key encryption system
  encrypts one secret message: a random 256-bit session key.

• Public-key signature system
  stops NSAITM attacks.

• Fast authenticated cipher
  uses the 256-bit session key to protect further messages.

Some cipher history

1973, and again in 1974:
U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism.
Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
**Introduction to symmetric crypto**

**D. J. Bernstein**

**How HTTPS protects connection:**

- Public-key encryption system encrypts one secret message: a random 256-bit session key.
- Public-key signature system stops NSAITM attacks.
- Fast authenticated cipher uses the 256-bit session key to protect further messages.

---

**Some cipher history**


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.
Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.
Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000 to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.

Some cipher history
1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.
1976: NSA meets Diffie and Hellman to discuss criticism. Claims "somewhere over $400,000,000" to break a DES key; "I don't think you can tell any Congressman what's going to be secure 25 years from now."
1977: DES is standardized.
1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
1978: Congressional investigation into NSA influence concludes "NSA convinced IBM that a reduced key size was sufficient".
Researchers publish new cipher proposals and security analysis.
1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1977: DES is standardized.
1977:Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.

Researchers publish new cipher proposals and security analysis.


1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.
1977: DES is standardized.

Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

A congressional investigation concludes "NSA convinced IBM that a key size was sufficient".


1998: EFF builds "Deep Crack" for under $250,000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES. "Security was the most important factor in the evaluation"—Really?
1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes "NSA convinced IBM that a reduced key size was sufficient".


Researchers publish new cipher proposals and security analysis.


1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

"Security was the most important factor in the evaluation"—Really?
1977: DES is standardized.
1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
1978: Congressional investigation into NSA influence concludes "NSA convinced IBM that a reduced key size was sufficient".
Researchers publish new cipher proposals and security analysis.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.
1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.
"Security was the most important factor in the evaluation"—Really?


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2019–now: NISTLWC competition.

5 AES proposals.

EFF builds “Deep Crack” for $250000 to break hundreds of DES keys per year.

NIST selects five finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2019–now: NISTLWC competition.

Main operations in AES:

\[ \text{add round key to block; apply substitution box } x \mapsto x^{254} \text{ in } F_{256} \text{ to each byte in block; linearly mix bits across block.} \]
National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.


1998: EFF builds “Deep Crack” for under $250,000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES. 

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. … Serpent appears to offer a high security margin.”


2019–now: NISTLWC competition.

Main operations in AES: add round key to block; apply substitution \( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \) to each byte in block; linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2019–now: NISTLWC competition.

Main operations in AES:
add round key to block; apply substitution box $x \mapsto x^{254}$ in $F_{256}$ to each byte in block;
linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”

2019–now: NISTLWC competition.

Main operations in AES:
add round key to block;
apply substitution box $x \mapsto x^{254}$ in $F_{256}$
to each byte in block;
linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . .
Serpent appears to offer a high security margin.”

2019–now: NISTLWC competition.

Main operations in AES:
add round key to block;
apply substitution box
\[ x \mapsto x^{254} \] in \( \mathbb{F}_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
Even in a post-quantum world, no serious threats to AES-256 in a strong security model, “multi-target SPRP security”.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”

2019–now: NISTLWC competition.

Main operations in AES: add round key to block; apply substitution box $x \mapsto x^{254}$ in $F_{256}$ to each byte in block; linearly mix bits across block.

Extensive security analysis. Even in a post-quantum world, no serious threats to AES-256 in a strong security model, “multi-target SPRP security”.

So why isn’t AES-256 the end of the symmetric-crypto story?
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

Serpent appears to offer an adequate security margin.

Rijndael appears to offer a high security margin.”

2008: eSTREAM competition for stream ciphers.

2012: SHA-3 competition.

2019: CAESAR competition.

Now: NISTLWC competition.

Main operations in AES:
add round key to block;
apply substitution box $x \mapsto x^{254}$ in $\mathbb{F}_{256}$
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
Even in a post-quantum world, no serious threats to AES-256 in a strong security model, “multi-target SPRP security”.

So why isn’t AES-256 the end of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
$x \mapsto x^{254}$ in $\mathbb{F}_{256}$
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
Even in a post-quantum world,
no serious threats to AES-256
in a strong security model,
“multi-target SPRP security”.

So why isn’t AES-256 the end
of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box \( x \mapsto x^{254} \) in \( F_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
Even in a post-quantum world,
no serious threats to AES-256
in a strong security model,
“multi-target SPRP security”.

So why isn’t AES-256 the end
of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
\( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
Even in a post-quantum world, no serious threats to AES-256 in a strong security model, “multi-target SPRP security”.

So why isn’t AES-256 the end of the symmetric-crypto story?
Main operations in AES:
- Add round key to block;
- Apply substitution box $x \rightarrow x^{254}$ in $F_{256}$ to each byte in block;
- Mix bits across block.

Extensive security analysis.
In a post-quantum world, no serious threats to AES-256 in a strong security model, "multi-target SPRP security".

So why isn't AES-256 the end of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
$x \mapsto x^{254}$ in $F_{256}$ to each byte in block;
linearly mix bits across block.

Extensive security analysis.

Even in a post-quantum world,
no serious threats to AES-256
in a strong security model,
"multi-target SPRP security".

So why isn't AES-256 the end
of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
$x \mapsto x^{254}$ in $F_{256}$
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
Even in a post-quantum world,
no serious threats to AES-256
in a strong security model,
"multi-target SPRP security".
So why isn't AES-256 the end
of the symmetric-crypto story?
acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms -- ChaCha 20 for symmetric encryption and Poly1305 for authentication -- in OpenSSL and NSS in March 2013. It was a complex effort that required implementing a new abstraction layer in OpenSSL in order to support the Authenticated Encryption with Associated Data (AEAD) encryption mode properly. AEAD enables encryption and authentication to happen concurrently, making it easier to use and optimize than older, commonly-used modes such as CBC. Moreover, recent attacks against RC4 and CBC also prompted us to make this change.

The benefits of this new cipher suite include:
acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms -- ChaCha 20 for symmetric encryption and Poly1305 for authentication -- in OpenSSL and NSS in March 2013. It was a complex effort that required implementing a new abstraction layer in OpenSSL in order to support the Authenticated Encryption with Associated Data (AEAD) encryption mode properly. AEAD enables encryption and authentication to happen concurrently, making it easier to use and optimize than older, commonly-used modes such as CBC. Moreover, recent attacks against RC4 and CBC also prompted us to make this change.

The benefits of this new cipher suite include:
Acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms -- ChaCha 20 for symmetric encryption and Poly1305 for authentication -- in OpenSSL and NSS in March 2013. It was a complex effort that required implementing a new abstraction layer in OpenSSL in order to support the Authenticated Encryption with Associated Data (AEAD) encryption mode properly. AEAD enables encryption and authentication to happen concurrently, making it easier to use and optimize than older, commonly-used modes such as CBC. Moreover, recent attacks against RC4 and CBC also prompted us to make this change.

The benefits of this new cipher suite include:

It was officially announced on the Cryptography mailing list [1]. While these devices still have to use older Cryptography Extensions for storage encryption, "Android Go" devices and others that support this new cipher suite still have to use older Cryptography Extensions.

As we explained in detail in the very strict paper describing the challenging problem of implementing AEAD in OpenSSL, the device is not suitable for practical use in the real world. We compared Speck, in this day and age, to a modern cipher there was a large political pressure.

Therefore, we (well, I did the encryption mode, Hal Finney wrote the ChaCha stream cipher code) published a paper here: https://
acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms -- ChaCha 20 for symmetric encryption and Poly1305 for authentication -- in OpenSSL and NSS in March 2013. It was a complex effort that required implementing a new abstraction layer in OpenSSL in order to support the Authenticated Encryption with Associated Data (AEAD) encryption mode properly. AEAD enables encryption and authentication to happen concurrently, making it easier to use and optimize than older, commonly-used modes such as CBC. Moreover, recent attacks against RC4 and CBC also prompted us to make this change.

The benefits of this new cipher suite include:

Date: 2018-08-06
Message-ID: 201808062233

From: Eric Biggers <ebiggers>

Hi all,

(Please note that this patch is still in progress and it to be merged quite yet!)

It was officially decided to remove support for RC4 and CBC encryption [1]. We've been using hardware-based storage encryption to entry-level "Android Go" devices sold in emerging markets, and these devices still ship with systems that have to use older CPUs like ARMv6 which do not have hardware AES support. The new cipher suite in OpenSSL also uses the Poly1305 AEAD mode. AEAD is more secure than AES-CBC and easier to use in the software implementation.

As we explained in detail earlier this year, the removal of CBC and the very strict performance requirements for entry-level devices makes Poly1305 the more suitable for practical use in those systems. We also considered Speck, in this day and age there's little to no competition for an advanced "light weight" cipher.

Therefore, we (well, Paul Curran and I) have changed the default AEAD encryption mode, HPolyC. In OpenSSL, this is achieved by using the ChaCha stream cipher for disk encryption and HPolyC for all other uses. To learn more about HPolyC, visit the paper here: https://eprint.iacr.org/2017/063.pdf
acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms -- ChaCha 20 for symmetric encryption and Poly1305 for authentication -- in OpenSSL and NSS in March 2013. It was a complex effort that required implementing a new abstraction layer in OpenSSL in order to support the Authenticated Encryption with Associated Data (AEAD) encryption mode properly. AEAD enables encryption and authentication to happen concurrently, making it easier to use and optimize than older, commonly-used modes such as CBC. Moreover, recent attacks against RC4 and CBC also prompted us to make this change.

The benefits of this new cipher suite include:

Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113899@bayes
[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a temporary stopgap and it to be merged quite yet!)

It was officially decided to *not* allow encryption [1]. We've been working to store encryption to entry-level Android "Android Go" devices sold in developing these devices still ship with no encryption. We have to use older CPUs like ARM Cortex-Cryptography Extensions, making AES-XTSC

As we explained in detail earlier, e.g., a challenging problem due to the lack of the very strict performance requirements suitable for practical use in dm-crypt. Speck, in this day and age the choice of has a large political element, restricting...

Therefore, we (well, Paul Crowley did the encryption mode, HPolyC. In essence, HPoly
ChaCha stream cipher for disk encryption paper here: https://eprint.iacr.org/2018
Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113891-1-ebiggers

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, it is unlikely that it will be merged quite yet!)

It was officially decided to *not* allow Android to encrypt data on boot for symmetric encryption and Poly1305 authentication -- in OpenSSL and NSS in March 2018. This is a complex effort that required adding a new abstraction layer in OpenSSL to support the Authenticated Encryption with Associated Data (AEAD) encryption mode properly. This enables encryption and authentication to be handled concurrently, making it easier to use and suitable for practical use in dm-crypt and fscrypt.

However, recent attacks against RC4 and CBC Speck, in this day and age the choice of cryptography has a large political element, restricting the options.

As we explained in detail earlier, e.g. in [2], the challenging problem due to the lack of encryption was the very strict performance requirements, while suitable for practical use in dm-crypt and fscrypt, AES-XTS with the ChaCha stream cipher for disk encryption. HPolyC [1] paper here: https://eprint.iacr.org/2018/720.pdf
Hi all,

(Please note that this patchset is a true RFC, i.e. we're not sure if it to be merged quite yet!)

It was officially decided to *not* allow Android devices to use encryption [1]. We've been working to find an alternative to storage encryption to entry-level Android devices like the "Android Go" devices sold in developing countries. Unfortunately, these devices still ship with no encryption, since for cost reasons have to use older CPUs like ARM Cortex-A7; and these CPUs lack Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a very challenging problem due to the lack of encryption algorithms and the very strict performance requirements, while still being suitable for practical use in dm-crypt and fscrypt. And as Speck, in this day and age the choice of cryptographic primitive has a large political element, restricting the options even further.

Therefore, we (well, Paul Crowley did the real work) designed another encryption mode, HPolyC. In essence, HPolyC makes it secure to use ChaCha stream cipher for disk encryption. HPolyC is specified in the PDF paper here: https://eprint.iacr.org/2018/720.pdf ("HPolyC: ...
Hi all,

(Please note that this patchset is a true RFC, i.e. we're not ready for it to be merged quite yet!)

It was officially decided to *not* allow Android devices to use Speck encryption [1]. We've been working to find an alternative way to bring storage encryption to entry-level Android devices like the inexpensive "Android Go" devices sold in developing countries. Unfortunately, once these devices still ship with no encryption, since for cost reasons they have to use older CPUs like ARM Cortex-A7; and these CPUs lack the AES-Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a very challenging problem due to the lack of encryption algorithms that meet the very strict performance requirements, while still being secure and suitable for practical use in dm-crypt and fscrypt. And as we saw with Speck, in this day and age the choice of cryptographic primitives also has a large political element, restricting the options even further.

Therefore, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it secure to use the ChaCha stream cipher for disk encryption. HPolyC is specified by our paper here: https://eprint.iacr.org/2018/720.pdf ("HPolyC:
Date: 2018-08-06 22:32:51
Message-ID: 20180806222300.113891-1-ebiggers () kernel ! org
[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.e. we're not ready for it to be merged quite yet!)

It was officially decided to *not* allow Android devices to use Speck encryption [1]. We've been working to find an alternative way to bring storage encryption to entry-level Android devices like the inexpensive "Android Go" devices sold in developing countries. Unfortunately, often these devices still ship with no encryption, since for cost reasons they have to use older CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8 Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a very challenging problem due to the lack of encryption algorithms that meet the very strict performance requirements, while still being secure and suitable for practical use in dm-crypt and fscrypt. And as we saw with Speck, in this day and age the choice of cryptographic primitives also has a large political element, restricting the options even further.

Therefore, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it secure to use the ChaCha stream cipher for disk encryption. HPolyC is specified by our paper here: https://eprint.iacr.org/2018/720.pdf ("HPolvC:
Eric Biggers <ebiggers@google.com>

Note that this patchset is a true RFC, i.e. we're not ready for
it to be merged quite yet!

Officially decided to *not* allow Android devices to use Speck
encryption [1]. We've been working to find an alternative way to bring
encryption to entry-level Android devices like the inexpensive
"Go" devices sold in developing countries. Unfortunately, often
these devices still ship with no encryption, since for cost reasons they
use older CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8
Technology Extensions, making AES-XTS much too slow.

As explained in detail earlier, e.g. in [2], this is a very
pressing problem due to the lack of encryption algorithms that meet
very strict performance requirements, while still being secure and
practical for use in dm-crypt and fscrypt. And as we saw with
at least one of our devices, other devices in this day and age the choice of cryptographic primitives also
become a political element, restricting the options even further.

Here, we (well, Paul Crowley did the real work) designed a new
stream cipher: HPolyC. In essence, HPolyC makes it secure to use the
stream cipher for disk encryption. HPolyC is specified by our
work:

https://eprint.iacr.org/2018/720.pdf ("HPolvC:
we decided to *not* allow Android devices to use Speck.

We've been working to find an alternative way to bring
speck to entry-level Android devices like the inexpensive
phones sold in developing countries. Unfortunately, often
they ship with no encryption, since for cost reasons they
use CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8
instructions, making AES-XTS much too slow.

In detail earlier, e.g. in [2], this is a very
problem due to the lack of encryption algorithms that meet
performance requirements, while still being secure and
suitable for use in dm-crypt and fscrypt. And as we saw with
the Baby and age the choice of cryptographic primitives also
becomes an element, restricting the options even further.

(Aside, Paul Crowley did the real work) designed a new
HPolyC. In essence, HPolyC makes it secure to use the
AES-XTS cipher for disk encryption. HPolyC is specified by our
paper at https://eprint.iacr.org/2018/720.pdf ("HPolvC:
The latest news and insights from Google on security on the Internet

Introducing Adiantum: Encryption for the Next Billion Users
February 7, 2019

Posted by Paul Crowley and Eric Biggers, Android Security Team

Privacy Team

Storage encryption protects your data if you...
true RFC, i.e. we're not ready for

How Android devices to use Speck
find an alternative way to bring
devices like the inexpensive
countries. Unfortunately, often
ption, since for cost reasons they
-A7; and these CPUs lack the ARMv8
S much too slow.

2, this is a very
encryption algorithms that meet
its, while still being secure and
and fscrypt. And as we saw with
of cryptographic primitives also
the options even further.

the real work) designed a new
HPolyC makes it secure to use the
on. HPolyC is specified by our
18/720.pdf ("HPolvC:
Introducing Adiantum: Encryption for the Next Billion Users

February 7, 2019

Posted by Paul Crowley and Eric Biggers, Android Security & Privacy Team

Storage encryption protects your data if your phone is stolen or lost. However, we're not ready for devices to use Speck encryption because it's slow.

This is a very young field of cryptography that is still being secure and fast. And as we saw with the ARMv8 architecture, it's expensive even further.

Our team (HPolvC) designed a new encryption scheme for the next billion users. HCTR and HCTR+ are split encryption modes that trade about 10.6% more encryption for AES-256-XTS.
Introducing Adiantum: Encryption for the Next Billion Users
February 7, 2019

The latest news and insights from Google on security and safety on the Internet

Introducing Adiantum: Encryption for the Next Billion Users
February 7, 2019

The latest news and insights from Google on security and safety on the Internet

Introducing Adiantum: Encryption for the Next Billion Users
February 7, 2019

The latest news and insights from Google on security and safety on the Internet
Introducing Adiantum: Encryption for the Next Billion Users
February 7, 2019

Posted by Paul Crowley and Eric Biggers, Android Security & Privacy Team

Storage encryption protects your data if your phone

Where AES is used, the conventional solution for encryption is to use the XTS or CBC-ESSIV encryption operation, which are length-preserving. Currently, Android supports AES-128-CBC-ESSIV for file-based encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient, there is no widely accepted alternative that has the same performance on lower-end ARM processors.

To solve this problem, we have designed and implemented a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from proposals for length-preserving encryption, such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
Introducing Adiantum: Encryption for the Next Billion Users
February 7, 2019

Posted by Paul Crowley and Eric Biggers, Android Security & Privacy Team

Storage encryption protects your data if your phone

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design.

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

Where AES is used, the conventional solution for disk encryption is to use the XTS or CBC-ESSIV modes of operation, which are length-preserving. Currently Android supports AES-128-CBC-ESSIV for full-disk encryption and AES-256-XTS for file-based encryption. However, when AES performance is insufficient there is no widely accepted alternative that has sufficient performance on lower-end ARM processors.

To solve this problem, we have designed a new encryption mode called Adiantum. Adiantum allows us to use the ChaCha stream cipher in a length-preserving mode, by adapting ideas from AES-based proposals for length-preserving encryption such as HCTR and HCH. On ARM Cortex-A7, Adiantum encryption and decryption on 4096-byte sectors is about 10.6 cycles per byte, around 5x faster than AES-256-XTS.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

Chacha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

Software ecosystem is complicated and dangerous. Software implementations of AES S-box often leak secrets through timing.

It is worse for high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets $r_1 \in \{0, 1, \ldots, 999999\}$, $r_2 \in \{0, 1, \ldots, 999999\}$, $r_5 \in \{0, 1, \ldots, 999999\}$, $s_1 \in \{0, 1, \ldots, 999999\}$, $s_{100} \in \{0, 1, \ldots, 999999\}$. 
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

System is dangerous. Implementations can leak timing.

For high-security authenticated ciphers. 128-bit block size limits “PRF” security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets $r_1 \in \{0, 1, \ldots, 999999\}$, $r_2 \in \{0, 1, \ldots, 999999\}$, $r_5 \in \{0, 1, \ldots, 999999\}$, $s_1 \in \{0, 1, \ldots, 999999\}$, $s_{100} \in \{0, 1, \ldots, 999999\}$.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy. AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing. Picture is worse for high-security authenticated ciphers. 128-bit block size limits "PRF" security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES. More examples of how symmetric primitives have been improving speed, simplicity, security: PRESENT is better than DES. Skinny is better than Simon and Speck. Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

Authentication details
Standardize a prime $p = 1000003$. Assume sender knows independent uniform random secrets $r_1 \in \{0, 1, \ldots, 999999\}$, $r_2 \in \{0, 1, \ldots, 999999\}$, $r_5 \in \{0, 1, \ldots, 999999\}$, $s_1 \in \{0, 1, \ldots, 999999\}$, $s_{100} \in \{0, 1, \ldots, 999999\}$. 
ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

Authentication details

Standardize a prime \( p = 1000003 \).

Assume sender knows independent uniform random secrets

\[ r_1 \in \{0, 1, \ldots, 999999\}, \]
\[ r_2 \in \{0, 1, \ldots, 999999\}, \]
\[ \vdots \]
\[ r_5 \in \{0, 1, \ldots, 999999\}, \]
\[ s_1 \in \{0, 1, \ldots, 999999\}, \]
\[ \vdots \]
\[ s_{100} \in \{0, 1, \ldots, 999999\}. \]
Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets

$r_1 \in \{0, 1, \ldots, 999999\}$,
$r_2 \in \{0, 1, \ldots, 999999\}$,
\[ \vdots \]
$r_5 \in \{0, 1, \ldots, 999999\}$,
$s_1 \in \{0, 1, \ldots, 999999\}$,
\[ \vdots \]
$s_{100} \in \{0, 1, \ldots, 999999\}$.

Assume receiver knows the same secrets $r_1; r_2; \ldots; r_5; s_1; \ldots; s_{100}$. 

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

- PRESENT is better than DES.
- Skinny is better than Simon and Speck.
- BLAKE2, Ascon are better than MD5, SHA-0, SHA-256, SHA-512.
Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets
$r_1 \in \{0, 1, \ldots, 999999\}$,
$r_2 \in \{0, 1, \ldots, 999999\}$,
$\vdots$
$r_5 \in \{0, 1, \ldots, 999999\}$,
$s_1 \in \{0, 1, \ldots, 999999\}$,
$\vdots$
$s_{100} \in \{0, 1, \ldots, 999999\}$.

Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$.
Authentication details

Standardize a prime \( p = 1000003 \).

Assume sender knows independent uniform random secrets
\( r_1 \in \{0, 1, \ldots, 999999\} \),
\( r_2 \in \{0, 1, \ldots, 999999\} \),
\( \vdots \)
\( r_5 \in \{0, 1, \ldots, 999999\} \),
\( s_1 \in \{0, 1, \ldots, 999999\} \),
\( \vdots \)
\( s_{100} \in \{0, 1, \ldots, 999999\} \).

Assume receiver knows the same secrets \( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100} \).
Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets $r_1 \in \{0, 1, \ldots, 999999\}$,
$r_2 \in \{0, 1, \ldots, 999999\}$,
\ldots,
$r_5 \in \{0, 1, \ldots, 999999\}$,
$s_1 \in \{0, 1, \ldots, 999999\}$,
\ldots,
$s_{100} \in \{0, 1, \ldots, 999999\}$.

Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$. 
Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets
$r_1 \in \{0, 1, \ldots, 999999\}$,
$r_2 \in \{0, 1, \ldots, 999999\}$,
\ldots
$r_5 \in \{0, 1, \ldots, 999999\}$,
s_1 \in \{0, 1, \ldots, 999999\}$,
\ldots
s_{100} \in \{0, 1, \ldots, 999999\}$.

Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0, 1, \ldots, 999999\}$.
Authentication details

Standardize a prime \( p = 1000003 \).

Assume sender knows independent uniform random secrets
\( r_1 \in \{0, 1, \ldots, 999999\} \),
\( r_2 \in \{0, 1, \ldots, 999999\} \),
\,
\( r_5 \in \{0, 1, \ldots, 999999\} \),
\( s_1 \in \{0, 1, \ldots, 999999\} \),
\,
\( s_{100} \in \{0, 1, \ldots, 999999\} \).

Assume receiver knows the same secrets \( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100} \).

Later: Sender wants to send 100 messages \( m_1, \ldots, m_{100} \),
each \( m_n \) having 5 components \( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)
with \( m_{n,i} \in \{0, 1, \ldots, 999999\} \).

Sender transmits 30-digit \( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)
together with an authenticator
\( (m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) + s_n \mod 1000000 \)
and the message number \( n \).
Authentication details

Standardize a prime $p = 1000003$.

Assume sender knows independent uniform random secrets $r_1 \in \{0, 1, \ldots, 999999\}$, $r_2 \in \{0, 1, \ldots, 999999\}$, $\ldots$, $r_5 \in \{0, 1, \ldots, 999999\}$, $s_1 \in \{0, 1, \ldots, 999999\}$, $\ldots$, $s_{100} \in \{0, 1, \ldots, 999999\}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0, 1, \ldots, 999999\}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ together with an authenticator $(m_{n,1} r_1 + \cdots + m_{n,5} r_5 \mod p) + s_n \mod 1000000$ and the message number $n$.

e.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006\ 000007\ 000000\ 000000\ 000000$.
Assume receiver knows the same secrets \( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100} \).

Later: Sender wants to send 100 messages \( m_1, \ldots, m_{100} \), each \( m_n \) having 5 components \( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \) with \( m_{n,i} \in \{0, 1, \ldots, 999999\} \).

Sender transmits 30-digit
\[
m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}
\]
together with an authenticator
\[
(m_{n,1} r_1 + \cdots + m_{n,5} r_5 \mod p) + s_n \mod 1000000
\]
and the message number \( n \).
Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0, 1, \ldots, 999999\}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ together with an authenticator $(m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) + s_n \mod 1000000$ and the message number $n$.

e.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006 000007 000000 000000 000000$.
Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0, 1, \ldots, 999999\}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ together with an authenticator $(m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) + s_n \mod 1000000$ and the message number $n$.

e.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006\ 000007\ 000000\ 000000\ 000000$: 
Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0, 1, \ldots, 999999\}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ together with an authenticator $\left( m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p \right) + s_n \mod 1000000$ and the message number $n$.

e.g. $r_1 = 314159$, $r_2 = 265358$,
$r_3 = 979323$, $r_4 = 846264$,
$r_5 = 338327$, $s_{10} = 950288$,
$m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000$:

Sender computes authenticator $(6r_1 + 7r_2 \mod p) + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 265358 \mod 1000003) + 950288 \mod 1000000 = 742451 + 950288 \mod 1000000 = 692739$. 
Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0, 1, \ldots, 999999\}$.

Sender computes authenticator $(6r_1 + 7r_2 \mod p)$
$+ s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 265358 \mod 1000003) + 950288 \mod 1000000 = 742451 + 950288 \mod 1000000 = 692739$.

Sender transmits
$10 000006 000007 000000 000000 000000 000000 000000 000000 000000 692739$. 

E.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006 000007 000000 000000 000000 000000 000000 000000 000000 000000 000000 692739$. 

Sender transmits $10 000006 000007 000000 000000 000000 000000 000000 000000 000000 692739$. 

Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$. Later: Sender wants to send $100$ messages $m_1, \ldots, m_{100}$, each $m_n$ having $5$ components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$, $i \in \{0, 1, \ldots, 999999\}$. Sender transmits $30$-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with an authenticator $(6r_1 + 7r_2 \mod p)$ + $s_n \mod 1000000$ + $\cdots + m_{n,5}r_5 \mod p) \mod 1000000$ and the message number $n$. A MAC using fewer secrets Instead of choosing independent $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$, choose $r, s_1, s_2, \ldots, s_{100}$.

E.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000$.

Sender computes authenticator $(6 \cdot 314159 + 7 \cdot 265358 \mod 1000003)$ + $950288 \mod 1000000 = 742451 + 950288 \mod 1000000 = 692739$.

Sender transmits $10 \ 000006 \ 000007 \ 000000 \ 000000 \ 000000 \ 692739$. 
Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5; s_1, \ldots, s_{100}$.

Later: Sender wants to send $100$ messages $m_1, \ldots, m_{100}$, each having $5$ components $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ with $m_{n,i} \in \{0,1,\ldots,999999\}$.

Sender transmits $30$-digit authenticator $(6r_1 + 7r_2 \mod p)$
\[+ s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 265358 \mod 1000003) + 950288 \mod 1000000 = 742451 + 950288 \mod 1000000 = 692739.\]

Sender transmits $1000006 000007 000000 000000 000000 692739$.

A MAC using fewer secrets

Instead of choosing independent $r_1, r_2, \ldots, r_5; s_1, \ldots, s_{100}$,
choose $r, s_1, s_2, \ldots, s_{100}$.

e.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006 000007 000000 000000 000000$:

Sender computes authenticator $(6r_1 + 7r_2 \mod p)$
\[+ s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 265358 \mod 1000003) + 950288 \mod 1000000 = 742451 + 950288 \mod 1000000 = 692739.\]
Assume receiver knows the same secrets $r_1, r_2, \ldots, r_5; s_1, \ldots, s_{100}$.

Later: Sender wants to send 100 messages $m_1, \ldots, m_{100}$, each $m_n$ having 5 components $m_n;_1, m_n;_2, m_n;_3, m_n;_4, m_n;_5$ with $m_n;i \in \{0,1,\ldots,999999\}$.

Sender transmits $30$-digit $m_n;_1, m_n;_2, m_n;_3, m_n;_4, m_n;_5$ together with an authenticator $(m_n;_1 r_1 + \cdots + m_n;_5 r_5 \mod p) + s_n \mod 1000000$ and the message number $n$.

e.g. $r_1 = 314159$, $r_2 = 265358$, $r_3 = 979323$, $r_4 = 846264$, $r_5 = 338327$, $s_{10} = 950288$, $m_{10} = 000006 000007 000000 000000 000000$:

Sender computes authenticator $(6r_1 + 7r_2 \mod p) + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 265358 \mod 1000003) + 950288 \mod 1000000 = 742451 + 950288 \mod 1000000 = 692739$.

Sender transmits $10 000006 000007 000000 000000 000000 692739$.

A MAC using fewer secrets
Instead of choosing independent $r_1, r_2, \ldots, r_5; s_1, \ldots, s_{100}$, choose $r, s_1, s_2, \ldots, s_{100}$. 
e.g. \( r_1 = 314159 \), \( r_2 = 265358 \),
\( r_3 = 979323 \), \( r_4 = 846264 \),
\( r_5 = 338327 \), \( s_{10} = 950288 \),
\( m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000 \ 000000 \).

Sender computes authenticator
\[
(6r_1 + 7r_2 \mod p)
+ s_{10} \mod 1000000 =
\]
\[
(6 \cdot 314159 + 7 \cdot 265358 \mod 1000003)
+ 950288 \mod 1000000 =
742451 + 950288 \mod 1000000 = 692739.
\]

Sender transmits
10 000006 000007 000000 000000 000000 692739.

A MAC using fewer secrets
Instead of choosing independent
\( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100} \),
choose \( r, s_1, s_2, \ldots, s_{100} \).
e.g. \( r_1 = 314159 \), \( r_2 = 265358 \),
\( r_3 = 979323 \), \( r_4 = 846264 \),
\( r_5 = 338327 \), \( s_{10} = 950288 \),
\( m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000 \):  

Sender computes authenticator
\[
(6r_1 + 7r_2 \mod p) \\
+ s_{10} \mod 1000000 =
\]
\[
(6 \cdot 314159 + 7 \cdot 265358 \mod 1000003) \\
+ 950288 \mod 1000000 =
\]
\[
742451 + 950288 \mod 1000000 = 692739.
\]

Sender transmits
10 000006 000007 000000 000000 000000 692739.

---

A MAC using fewer secrets

Instead of choosing independent
\( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100} \),
choose \( r, s_1, s_2, \ldots, s_{100} \).

Sender transmits 30-digit
\( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)
together with an authenticator
\[
( m_{n,1}r + \cdots + m_{n,5}r^5 \mod p ) \\
+ s_n \mod 1000000
\]
and the message number \( n \).

i.e.: take \( r_i = r^i \) in previous
\[
( m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p ) \\
+ s_n \mod 1000000.
\]
A MAC using fewer secrets

Instead of choosing independent 
\( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}, \)
choose \( r, s_1, s_2, \ldots, s_{100}. \)

Sender transmits 30-digit
\( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)
together with an authenticator 
\((m_{n,1}r + \cdots + m_{n,5}r^5 \mod p)\)
\(+ s_n \mod 1000000\)
and the message number \( n. \)

i.e. take \( r_i = r^i \) in previous
\((m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p)\)
\(+ s_n \mod 1000000. \)
A MAC using fewer secrets

Instead of choosing independent $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$, choose $r, s_1, s_2, \ldots, s_{100}$.

Sender transmits 30-digit

$m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$

together with an authenticator

$(m_{n,1}r + \cdots + m_{n,5}r^5 \mod p) + s_n \mod 1000000$

and the message number $n$.

i.e.: take $r_i = r^i$ in previous

$(m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) + s_n \mod 1000000$. 

\[ r_2 = 265358, \quad r_3 = 950288, \quad m_{10} = 000006\ldots \]
A MAC using fewer secrets

Instead of choosing independent $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$, choose $r, s_1, s_2, \ldots, s_{100}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ together with an authenticator $(m_{n,1} r + \cdots + m_{n,5} r^5 \mod p) + s_n \mod 1000000$
and the message number $n$.

i.e.: take $r_i = r^i$ in previous $(m_{n,1} r_1 + \cdots + m_{n,5} r_5 \mod p) + s_n \mod 1000000$.
A MAC using fewer secrets

Instead of choosing independent 
\( r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100} \),
choose \( r, s_1, s_2, \ldots, s_{100} \).

Sender transmits 30-digit 
\( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)

\( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)
together with an authenticator
\((m_{n,1}r + \cdots + m_{n,5}r^5 \mod p) \)
\( + s_n \mod 1000000 \)

and the message number \( n \).

i.e.: take \( r_i = r^i \) in previous 
\((m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) \)
\( + s_n \mod 1000000 \).

\text{e.g.} \, r = 314159, \, s_{10} = 265358, \\
m_{10} = 000006 \, 000007 \, 000000 \, 000000 \, 000000 :
A MAC using fewer secrets

Instead of choosing independent $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$,
choose $r, s_1, s_2, \ldots, s_{100}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$
together with an authenticator $(m_{n,1}r + \cdots + m_{n,5}r^5 \mod p)$
$+ s_n \mod 1000000$
and the message number $n$.

i.e.: take $r_i = r^i$ in previous $(m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p)$
$+ s_n \mod 1000000$.

e.g. $r = 314159$, $s_{10} = 265358$, $m_{10} = 000006 000007 000000 000000 000000$: Sender computes authenticator $(6r + 7r^2 \mod p)$
$+ s_10 \mod 1000000 =$
$(6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003)$
$+ 265358 \mod 1000000 =$
$953311 + 265358 \mod 1000000 = 218669$. 

[Note: The above text is a cryptographic protocol for message authentication code (MAC) using fewer secrets. It describes how to compute a MAC with fewer secrets than traditional methods, and includes an example calculation.]
A MAC using fewer secrets

Instead of choosing independent $r_1, r_2, \ldots, r_5, s_1, \ldots, s_{100}$, choose $r, s_1, s_2, \ldots, s_{100}$.

Sender transmits 30-digit $m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5}$ together with an authenticator $(m_{n,1}r + \cdots + m_{n,5}r^5 \mod p) + s_n \mod 1000000$
and the message number $n$.

i.e.: take $r_i = r^i$ in previous $(m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) + s_n \mod 1000000$.

E.g. $r = 314159$, $s_{10} = 265358$, $m_{10} = 000006\ 000007\ 000000\ 000000\ 000000\ 000000$:

Sender computes authenticator $(6r + 7r^2 \mod p)$
$+ s_{10} \mod 1000000 =$
$(6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003) + 265358 \mod 1000000 =$
$953311 + 265358 \mod 1000000 = 218669$.

Sender transmits authenticated message $10\ 000006\ 000007\ 000000\ 000000\ 000000\ 000000\ 000000\ 000000\ 218669$. 

A MAC using fewer secrets

Instead of choosing independent \( r_1, r_2, \ldots, r_5, s_1, s_2, \ldots, s_{100}, \)
choose \( r, s_1, s_2, \ldots, s_{100}. \)

Sender transmits 30-digit \( m_{n,1}, m_{n,2}, m_{n,3}, m_{n,4}, m_{n,5} \)
with an authenticator \( (m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) + s_1 \mod 1000000 \)
and the message number \( n. \)

i.e.: take \( r_i = r^i \) in previous \( (m_{n,1}r_1 + \cdots + m_{n,5}r_5 \mod p) \) mod 1000000.

e.g. \( r = 314159, \ s_{10} = 265358, \)
\( m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000: \)

Sender computes authenticator \( (6r + 7r^2 \mod p) \)
\( + s_{10} \mod 1000000 = \)
\( (6 \cdot 314159 + 7 \cdot 314159^2 \)
\( \mod 1000003) \)
\( + 265358 \mod 1000000 = \)
\( 953311 + 265358 \mod 1000000 = \)
\( 218669. \)

Sender transmits authenticated message
10 000006 000007 000000 000000 000000 218669.

Security analysis

Attacker's goal:
Find \( n', m', a' \) such that \( m' \neq m_n \) but \( a' = (m'(r) \mod p) + s_{n'} \mod 1000000. \)
Here \( m'(x) = \sum_{i} m'_{[i]} x^i. \)
A MAC using fewer secrets

Instead of choosing independent $r_1, r_2, \ldots, r_5; s_1, \ldots, s_{100}$, choose $r, s_1, s_2, \ldots, s_{100}$.

Sender transmits 30-digit $m_{10} = 000006 00007 00000 00000 00000 00000$, together with an authenticator

$\left( m_{10} r_1 + \cdots + m_{10} r_5 \mod p \right) + s_{10} \mod 1000000$.

i.e.: take $r_i = r_i$ in previous

$\left( m_n r_1 + \cdots + m_n r_5 \mod p \right) + s_n \mod 1000000$.

e.g. $r = 314159$, $s_{10} = 265358$

$m_{10} = 000006 00007 00000 00000 00000 00000$:

Sender computes authenticator

$\left( 6r + 7r^2 \mod p \right) + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003)$

$+ 265358 \mod 1000000 = 953311 + 265358 \mod 1000000 = 218669$.

Sender transmits authenticated message

$10 000006 00007 00000 00000 00000 00000 218669$.

Security analysis

Attacker’s goal:

Find $n', m', a'$ such that $m' \neq m_{n'}$ but $a' = (m'(r) \mod p) + s_{n'} \mod 1000000$.

Here $m'(x) = \sum_i m'_{i} x_{i}$.
A MAC using fewer secrets

Instead of choosing independent $r_1; r_2; \ldots; r_5; s_1; \ldots; s_{100}$, choose $r; s_1; s_2; \ldots; s_{100}$.

Sender transmits 30-digit $m_1; m_2; m_3; m_4; m_5$ together with an authenticator $(m_1 r_1 + \cdots + m_5 r_5 \mod p) + s_n \mod 1000000$ and the message number $n$.

e.g. $r = 314159, s_{10} = 265358, m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000$:

Sender computes authenticator $(6r + 7r^2 \mod p)$

$\quad + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003)$

$\quad + 265358 \mod 1000000 = 953311 + 265358 \mod 1000000 = 218669.$

Sender transmits authenticated message

$10 \ 000006 \ 000007 \ 000000 \ 000000 \ 000000 \ 000000 \ 218669$.

Security analysis

Attacker’s goal: Find $n', m', a'$ such that $m' \neq m_{n'}$ but $a' = (m'(r) \mod p) + s_{n'} \mod 1000000$.

Here $m'(x) = \sum_i m'[i]x^i$. 
e.g. $r = 314159$, $s_{10} = 265358$, $m_{10} = 000006\ 000007\ 000000\ 000000\ 000000\ 000000\ 000000\ 000000$:  

Sender computes authenticator 

$$(6r + 7r^2 \mod p) + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003) + 265358 \mod 1000000 = 953311 + 265358 \mod 1000000 = 218669.$$

Sender transmits 

authenticated message $10\ 000006\ 000007\ 000000\ 000000\ 000000\ 000000\ 218669$.  

Security analysis

Attacker’s goal: 

Find $n'$, $m'$, $a'$ such that $m' \neq m_{n'}$ but $a' = (m'(r) \mod p) + s_{n'} \mod 1000000$. 

Here $m'(x) = \sum_i m'[i]x^i$. 
e.g. $r = 314159$, $s_{10} = 265358$, $m_{10} = 000006 000007 000000 000000 000000 000000 000000$:

Sender computes authenticator

$$(6r + 7r^2 \mod p) + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003) + 265358 \mod 1000000 = 953311 + 265358 \mod 1000000 = 218669.$$ 

Sender transmits authenticated message

10 000006 000007 000000 000000 000000 000000 218669.

Security analysis

Attacker's goal:
Find $n', m', a'$ such that
$m' \neq m_{n'}$ but $a' = (m'(r) \mod p) + s_{n'} \mod 1000000$.
Here $m'(x) = \sum_i m'[i]x^i$.

Obvious attack:
Choose any $m' \neq m_1$.
Choose uniform random $a'$.
Success chance $1/1000000$. 
e.g. \( r = 314159 \), \( s_{10} = 265358 \),
\( m_{10} = 000006 \ 000007 \ 000000 \ 000000 \ 000000 \ 000000 \) :

Sender computes authenticator
\((6r + 7r^2 \mod p) + s_{10} \mod 1000000 = (6 \cdot 314159 + 7 \cdot 314159^2 \mod 1000003) + 265358 \mod 1000000 = 953311 + 265358 \mod 1000000 = 218669.\)

Sender transmits authenticated message
\( 10 \ 000006 \ 000007 \ 000000 \ 000000 \ 000000 \ 218669. \)

**Security analysis**

Attacker’s goal:
Find \( n', m', a' \) such that
\( m' \neq m_{n'} \) but \( a' = (m'(r) \mod p) + s_{n'} \mod 1000000. \)

Here \( m'(x) = \sum_i m'[i]x^i. \)

Obvious attack:
Choose any \( m' \neq m_1. \)
Choose uniform random \( a'. \)
Success chance \( 1/1000000. \)

Can repeat attack.
Each forgery has chance \( 1/1000000 \) of being accepted.
\[ r = 314159, \quad s_{10} = 265358, \quad m_{10} = 000006\ 000007\ 000000\ 000000\ 000000\ : \]

Sender computes authenticator
\[ (6 \cdot 314159 + 7 \cdot 314159^2 \mod p) \]
\[ + 265358 \mod 1000000 = 953311 + 265358 \mod 1000000 = 218669. \]

Transmits authenticated message \( 000006\ 000007\ 000000\ 000000\ 000000\ 218669. \)

**Security analysis**

Attacker’s goal:
Find \( n', m', a' \) such that \( m' \neq m_n \) but \( a' = (m'(r) \mod p) + s_{n'} \mod 1000000. \)

Here \( m'(x) = \sum_i m'[i]x^i. \)

Obvious attack:
Choose any \( m' \neq m_1. \)
Choose uniform random \( a'. \)
Success chance \( 1/1000000. \)

Can repeat attack.
Each forgery has chance \( 1/1000000 \) of being accepted.

More subtle attack:
Choose \( m' \neq m_1 \) so that the polynomial \( m'(x) \square m_1(x) \)
has 5 distinct roots \( x \in \{0, 1, \ldots, 999999\} \) modulo \( p. \)
Choose \( a' = a. \)
Security analysis

Attacker’s goal:
Find \( n' \), \( m' \), \( a' \) such that
\( m' \neq m_{n'} \) but \( a' = (m'(r) \mod p) + s_{n'} \mod 1000000 \).
Here \( m'(x) = \sum_i m'[i]x^i \).

Obvious attack:
Choose any \( m' \neq m_1 \).
Choose uniform random \( a' \).
Success chance \( 1/1000000 \).

Can repeat attack.
Each forgery has chance
\( 1/1000000 \) of being accepted.

More subtle attack:
Choose \( m' \neq m_1 \) so that the polynomial \( m'(x) \)
has 5 distinct roots \( x \in \{0, 1, \ldots, 999999\} \mod p \).
Choose \( a' = a \).
Security analysis

Attacker’s goal:
Find \( n', m', a' \) such that
\( m' \neq m_{n'} \) but \( a' = (m'(r) \mod p) + s_{n'} \mod 1000000 \).
Here \( m'(x) = \sum_i m'[i]x^i \).

Obvious attack:
Choose any \( m' \neq m_1 \).
Choose uniform random \( a' \).
Success chance \( 1/1000000 \).

Can repeat attack.
Each forgery has chance \( 1/1000000 \) of being accepted.

More subtle attack:
Choose \( m' \neq m_1 \) so that the polynomial \( m'(x) - m_1(x) \) has 5 distinct roots \( x \in \{0, 1, \ldots, 999999\} \mod p \). Choose \( a' = a \).
Security analysis

Attacker’s goal:
Find \( n', m', a' \) such that 

\[ m' \neq m_n' \text{ but } a' = (m'(r) \mod p) + s_{n'} \mod 1000000. \]

Here \( m'(x) = \sum_i m'[i]x^i. \)

Obvious attack:
Choose any \( m' \neq m_1. \)
Choose uniform random \( a'. \)
Success chance 1/1000000.

Can repeat attack.
Each forgery has chance 
\[ 1/1000000 \] of being accepted.

More subtle attack:
Choose \( m' \neq m_1 \) so that 
the polynomial \( m'(x) - m_1(x) \)
has 5 distinct roots 
\( x \in \{0, 1, \ldots, 999999\} \)
modulo \( p. \) Choose \( a' = a. \)
Security analysis

Attacker’s goal:
Find $n', m', a'$ such that $m' \neq m_n'$ but $a' = (m'(r) \mod p) + s_{n'} \mod 1000000$.
Here $m'(x) = \sum_i m'[i]x^i$.

Obvious attack:
Choose any $m' \neq m_1$.
Choose uniform random $a'$.
Success chance $1/1000000$.

Can repeat attack.
Each forgery has chance $1/1000000$ of being accepted.

More subtle attack:
Choose $m' \neq m_1$ so that the polynomial $m'(x) - m_1(x)$ has 5 distinct roots
$x \in \{0, 1, \ldots, 999999\}$ modulo $p$. Choose $a' = a$.

e.g. $m_1 = (100, 0, 0, 0, 0)$,
$m' = (125, 1, 0, 0, 1)$:
$m'(x) - m_1(x) = x^5 + x^2 + 25x$
which has five roots mod $p$:
0, 299012, 334447, 631403, 735144.
Security analysis

Attacker’s goal:
Find \( n', m', a' \) such that
\( m' \neq m_n' \) but \( a' = (m'(r) \mod p) + s_{n'} \mod 1000000. \)
Here \( m'(x) = \sum_i m'[i]x^i. \)

Obvious attack:
Choose any \( m' \neq m_1. \)
Choose uniform random \( a'. \)
Success chance \( 1/1000000. \)

Can repeat attack.
Each forgery has chance \( 1/1000000 \) of being accepted.

More subtle attack:
Choose \( m' \neq m_1 \) so that
the polynomial \( m'(x) - m_1(x) \)
has 5 distinct roots
\( x \in \{0, 1, \ldots, 999999\} \)
modulo \( p. \) Choose \( a' = a. \)

\( e.g. \ m_1 = (100, 0, 0, 0, 0), \)
\( m' = (125, 1, 0, 0, 1): \)
\( m'(x) - m_1(x) = x^5 + x^2 + 25x \)
which has five roots mod \( p: \)
0, 299012, 334447, 631403, 735144.

Success chance \( 5/1000000. \)
More subtle attack:
Choose \( m' \neq m_1 \) so that
the polynomial \( m'(x) - m_1(x) \)
has 5 distinct roots
\( x \in \{0, 1, \ldots, 999999\} \)
modulo \( p \). Choose \( a' = a \).

\[
\begin{align*}
\text{e.g. } & m_1 = (100, 0, 0, 0, 0), \\
& m' = (125, 1, 0, 0, 1): \\
& m'(x) - m_1(x) = x^5 + x^2 + 25x \\
\end{align*}
\]
which has five roots mod \( p \):
0, 299012, 334447, 631403, 735144.

Success chance 5/1000000.
Security analysis

Attacker's goal:
Find $n'$; $m'$; $a'$ such that $m' \neq m^{n'}$ but $a' = (m'(r) \mod p) + s^{n'} \mod 1000000$.

Here $m'(x) = \sum_{i} m'[i]x^i$.

Obvious attack:
Choose any $m' \neq m_1$.
Choose uniform random $a'$.
Success chance $1 = 1000000$.
Can repeat attack.
Each forgery has chance $1 = 1000000$ of being accepted.

More subtle attack:
Choose $m' \neq m_1$ so that the polynomial $m'(x) - m_1(x)$ has 5 distinct roots $x \in \{0, 1, \ldots, 999999\}$ modulo $p$. Choose $a' = a$.

e.g. $m_1 = (100, 0, 0, 0, 0)$,
$m' = (125, 1, 0, 0, 1)$:
$m'(x) - m_1(x) = x^5 + x^2 + 25x$
which has five roots mod $p$:
0, 299012, 334447, 631403, 735144.

Success chance $5/1000000$.

Actually, success chance can be above $5/1000000$. 

More subtle attack:
Choose $m' \neq m_1$ so that
the polynomial $m'(x) - m_1(x)$
has 5 distinct roots
$x \in \{0, 1, \ldots, 999999\}$
modulo $p$. Choose $a' = a$.

e.g. $m_1 = (100, 0, 0, 0, 0),$
$m' = (125, 1, 0, 0, 1):$
$m'(x) - m_1(x) = x^5 + x^2 + 25x$
which has five roots mod $p$:
0, 299012, 334447, 631403, 735144.

Success chance $5/1000000$.

Actually, success chance
can be above $5/1000000$. 
More subtle attack:
Choose \( m' \neq m_1 \) so that the polynomial \( m'(x) - m_1(x) \) has 5 distinct roots \( x \in \{0, 1, \ldots, 999999\} \) modulo \( p \). Choose \( a' = a \).

e.g. \( m_1 = (100, 0, 0, 0, 0) \), \( m' = (125, 1, 0, 0, 1) \):

\[
m'(x) - m_1(x) = x^5 + x^2 + 25x
\]

which has five roots mod \( p \): 0, 299012, 334447, 631403, 735144.

Success chance \( 5/1000000 \).

Actually, success chance can be above \( 5/1000000 \).
More subtle attack:
Choose $m' \neq m_1$ so that
the polynomial $m'(x) - m_1(x)$
has 5 distinct roots
$x \in \{0, 1, \ldots, 999999\}$
modulo $p$. Choose $a' = a$.

e.g. $m_1 = (100, 0, 0, 0, 0)$,
$m' = (125, 1, 0, 0, 1)$:
$m'(x) - m_1(x) = x^5 + x^2 + 25x$
which has five roots mod $p$:
0, 299012, 334447, 631403, 735144.

Success chance 5/1000000.

Actually, success chance can be above 5/1000000.

Example: If $m_1(334885) \mod p 
\in \{1000000, 1000001, 1000002\}$
then a forgery $(1, m', a_1)$ with
$m'(x) = m_1(x) + x^5 + x^2 + 25x$
also succeeds for $r = 334885$;
success chance 6/1000000.

Reason: 334885 is a root of
$m'(x) - m_1(x) + 1000000.$
More subtle attack:
Choose $m' \neq m_1$ so that
the polynomial $m'(x) - m_1(x)$
has 5 distinct roots
$x \in \{0, 1, \ldots, 999999\}$
modulo $p$. Choose $a' = a$.

e.g. $m_1 = (100, 0, 0, 0, 0)$,
$m' = (125, 1, 0, 0, 1)$:
$m'(x) - m_1(x) = x^5 + x^2 + 25x$
which has five roots mod $p$:
0, 299012, 334447, 631403, 735144.

Success chance 5/1000000.

Actually, success chance can be above 5/1000000.

Example: If $m_1(334885) \mod p$
$\in \{1000000, 1000001, 1000002\}$
then a forgery $(1, m', a_1)$ with
$m'(x) = m_1(x) + x^5 + x^2 + 25x$
also succeeds for $r = 334885$;
success chance 6/1000000.
Reason: 334885 is a root of
$m'(x) - m_1(x) + 1000000$.

Can have as many as 15 roots
of $(m'(x) - m_1(x)) \cdot$
$(m'(x) - m_1(x) + 1000000) \cdot$
$(m'(x) - m_1(x) - 1000000)$. 
More subtle attack:
Choose $m' \neq m_1$ so that
the polynomial $m'(x) - m_1(x)$ has 5 distinct roots
$x \in \{0; 1; \ldots; 999999\}$ modulo $p$. Choose $a' = a$.

$m_1(x) = x^5 + x^2 + 25x$
as a polynomial with 5 roots modulo $p$:
$0, 299012, 334447, 631403, 735144$.

Success chance 5 = 1/1000000.

Actually, success chance can be above 5/1000000.

Example: If $m_1(334885) \mod p \in \{1000000, 1000001, 1000002\}$
then a forgery $(1, m', a_1)$ with
$m'(x) = m_1(x) + x^5 + x^2 + 25x$
also succeeds for $r = 334885$; success chance 6/1000000.

Reason: 334885 is a root of $m'(x) - m_1(x) + 1000000$.

Can have as many as 15 roots
of $(m'(x) - m_1(x)) \cdot (m'(x) - m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000)$.

Do better by varying $a'$?
More subtle attack:
Choose \( m' \neq m_1 \) so that the polynomial \( m'(x) \cdot m_1(x) \) has 5 distinct roots
\( x \in \{ \ldots, 0, 1, \ldots, 999999 \} \) modulo \( p \).
Choose \( a' = a \).
Example: If \( m_1(334885) \) mod \( p \) \in \{ 1000000, 1000001, 1000002 \} then a forgery \((1, m', a_1)\) with \( m'(x) = m_1(x) + x^5 + x^2 + 25x \) also succeeds for \( r = 334885 \); success chance 6/1000000.
Reason: 334885 is a root of \( m'(x) - m_1(x) + 1000000 \).
Can have as many as 15 roots of \(( m'(x) - m_1(x) \) \cdot \( m'(x) - m_1(x) + 1000000 \) \cdot \( m'(x) - m_1(x) - 1000000 \).

Actually, success chance can be above 5/1000000.

Example: If \( m_1(334885) \) mod \( p \) \in \{ 1000000, 1000001, 1000002 \} then a forgery \((1, m', a_1)\) with \( m'(x) = m_1(x) + x^5 + x^2 + 25x \) also succeeds for \( r = 334885 \); success chance 6/1000000.
Reason: 334885 is a root of \( m'(x) - m_1(x) + 1000000 \).
Can have as many as 15 roots of \(( m'(x) - m_1(x) \) \cdot \( m'(x) - m_1(x) + 1000000 \) \cdot \( m'(x) - m_1(x) - 1000000 \).

Do better by varying \( a' \)?
More subtle attack:
Choose \( m' \neq m_1 \) so that the polynomial \( m'(x) \mod m_1(x) \) has 5 distinct roots \( x \in \{0; 1; \ldots; 999999\} \) modulo \( p \). Choose \( a' = a_1 \).

\[ m'(x) = m_1(x) + x^5 + x^2 + 25x \]
which has five roots mod \( p \):
\[ 0; 299012; 334447; 631403; 735144. \]

Success chance \( 5 = 1000000 \).

Actually, success chance can be above \( 5 = 1000000 \).
Example: If \( m_1(334885) \mod p \in \{1000000, 1000001, 1000002\} \) then a forgery \((1, m', a_1)\) with
\[ m'(x) = m_1(x) + x^5 + x^2 + 25x \]
also succeeds for \( r = 334885 \);
success chance \( 6/1000000 \).

Reason: \( 334885 \) is a root of
\[ m'(x) - m_1(x) + 1000000. \]
Can have as many as 15 roots of
\( (m'(x) - m_1(x)) \cdot (m'(x) - m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000). \)

Do better by varying \( a' \)?
Actually, success chance can be above \( 5/1000000 \).

Example: If \( m_1(334885) \mod p \in \{1000000, 1000001, 1000002\} \) then a forgery \((1, m', a_1)\) with \( m'(x) = m_1(x) + x^5 + x^2 + 25x \) also succeeds for \( r = 334885\); success chance \( 6/1000000 \).

Reason: 334885 is a root of \( m'(x) - m_1(x) + 1000000 \).

Can have as many as 15 roots of \( (m'(x) - m_1(x)) \cdot (m'(x) - m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000) \).
Actually, success chance can be above $5/1000000$.

Example: If $m_1(334885) \mod p \in \{1000000, 1000001, 1000002\}$ then a forgery $(1, m', a_1)$ with $m'(x) = m_1(x) + x^5 + x^2 + 25x$ also succeeds for $r = 334885$; success chance $6/1000000$.

Reason: $334885$ is a root of $m'(x) - m_1(x) + 1000000$.

Can have as many as $15$ roots of $(m'(x) - m_1(x)) \cdot (m'(x) + m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000)$.

Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_{n'}$ has chance $\leq 15/1000000$ of being accepted by receiver.
Actually, success chance can be above $5/1000000$.

Example: If $m_1(334885) \mod p \in \{1000000, 1000001, 1000002\}$ then a forgery $(1, m', a_1)$ with $m'(x) = m_1(x) + x^5 + x^2 + 25x$ also succeeds for $r = 334885$; success chance $6/1000000$.

Reason: 334885 is a root of $m'(x) - m_1(x) + 1000000$.

Can have as many as 15 roots of $(m'(x) - m_1(x)) \cdot (m'(x) - m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000)$.

Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_{n'}$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6)$.
Actually, success chance can be above \(5/1000000\).

Example: If \(m_1(334885) \mod p \in \{1000000, 1000001, 1000002\}\) then a forgery \((1, m', a_1)\) with 
\[m'(x) = m_1(x) + x^5 + x^2 + 25x\]
also succeeds for \(r = 334885\); success chance \(6/1000000\).

Reason: 334885 is a root of 
\[(m'(x) - m_1(x)) \cdot (m'(x) - m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000).\]

Can have as many as 15 roots of \((m'(x) - m_1(x)) \cdot (m'(x) - m_1(x) + 1000000) \cdot (m'(x) - m_1(x) - 1000000).\)

Do better by varying \(a'\)?

No. Easy to prove: Every choice of \((n', m', a')\) with \(m' \neq m_n\)
has chance \(\leq 15/1000000\) of being accepted by receiver.

Underlying fact: \(\leq 15\) roots of 
\[(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6).\]

Warning: very easy to break the oversimplified authenticator 
\[(m_n[1] + \cdots + m_n[5] r^4 \mod p) + s_n \mod 1000000: \]
solve \(m'(x) - m_1(x) = a' - a_1.\)
Actually, success chance can be above $5/1000000$.

Example: If $m_1(334885) \mod p \in \{1000000, 1000001, 1000002\}$ Borgery $(1, m', a_1)$ with
$m_1(x) + x^5 + x^2 + 25x$ succeeds for $r = 334885$; chance $6/1000000$.

$334885$ is a root of $m_1(x) + 1000000$.

Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_{n'}$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6)$.

Warning: very easy to break the oversimplified authenticator $(m_n[1] + \cdots + m_n[5]r^4 \mod p) + s_n \mod 1000000:
\text{solve } m'(x) - m_1(x) = a' - a_1$.

Scaled up for serious security:
Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed.
Adds $s_n \mod 2^{128}$. 
Actually, success chance can be above 5 \(= 1000000\).

Example: If \(m_1(334885) \mod p \in \{1000000; 1000001; 1000002\}\) then a forgery \((m'; m_1'; a_1')\) with
\[m'(x) = m_1(x) + x^5 + x^2 + 25x^3 + x + 25 + x^2 + 25x^3 + x + 25 + x\]
also succeeds for \(r = 334885\); success chance \(\leq \frac{15}{1000000}\).

Reason: \(334885\) is a root of \((m'(x) - m_1(x) - a' + a_1)\) · 
\((m'(x) - m_1(x) - a' + a_1 + 10^6)\) · 
\((m'(x) - m_1(x) - a' + a_1 - 10^6)\).

Warning: very easy to break the oversimplified authenticator
\((m_n[1] + \cdots + m_n[5] r^4 \mod p) + s_n \mod 1000000:\)

solve \(m'(x) - m_1(x) = a' - a_1\).

Do better by varying \(a'\)?

No. Easy to prove: Every choice of \((n', m', a')\) with \(m' \neq m_n'\) has chance \(\leq \frac{15}{1000000}\) of being accepted by receiver.

Underlying fact: \(\leq 15\) roots
of \((m'(x) - m_1(x) - a' + a_1)\) · 
\((m'(x) - m_1(x) - a' + a_1 + 10^6)\) · 
\((m'(x) - m_1(x) - a' + a_1 - 10^6)\).

Scaled up for serious security:
Poly1305 uses 128-bit \(r\)'s, with 22 bits cleared for speed.
 Adds \(s_n \mod 2^{128}\).
Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_{n'}$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of 

$$(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6).$$

Warning: very easy to break the oversimplified authenticator 

$$(m_n[1] + \cdots + m_n[5]r^4 \mod p) + s_n \mod 1000000:$$

solve $m'(x) - m_1(x) = a' - a_1$.

Scaled up for serious security: Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.
Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_n'$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of 

$$(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6).$$

Warning: very easy to break the oversimplified authenticator 

$$(m_n[1] + \cdots + m_n[5]r^4 \mod p) + s_n \mod 1000000:$$

solve $m'(x) - m_1(x) = a' - a_1$. 

Scaled up for serious security: 
Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$. 
Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_n'$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6)$.

Warning: very easy to break the oversimplified authenticator $(m_n[1] + \cdots + m_n[5] r^4 \mod p) + s_n \mod 1000000$:

solve $m'(x) - m_1(x) = a' - a_1$.

Scaled up for serious security: Poly1305 uses 128-bit $r$’s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.

Assuming $\leq L$-byte messages:
Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$.
Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$. 
Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_n'$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6)$.

Warning: very easy to break the oversimplified authenticator $(m_n[1] + \cdots + m_n[5]r^4 \mod p) + s_n \mod 1000000$: solve $m'(x) - m_1(x) = a' - a_1$.

Scaled up for serious security:

Poly1305 uses 128-bit $r'$s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.

Assuming $\leq L$-byte messages:

Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$.

Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lceil L/16 \rceil / 2^{106}$. 

Do better by varying $a'$?

No. Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_n'$ has chance $\leq 15/1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x) - m_1(x) - a' + a_1) \cdot (m'(x) - m_1(x) - a' + a_1 + 10^6) \cdot (m'(x) - m_1(x) - a' + a_1 - 10^6)$.

Warning: very easy to break the oversimplified authenticator $(m_n[1] + \cdots + m_n[5] r^4 \text{ mod } p) + s_n \text{ mod } 1000000$:
solve $m'(x) - m_1(x) = a' - a_1$.

Scaled up for serious security:

Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed.
Adds $s_n \text{ mod } 2^{128}$.

Assuming $\leq L$-byte messages:
Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$.
Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lceil L/16 \rceil / 2^{106}$.

E.g. $2^{64}$ forgeries, $L = 1536$:
$\Pr[\text{all rejected}] \geq 0.9999999998$. 
Do better by varying $a'$?

Easy to prove: Every choice of $(n', m', a')$ with $m' \neq m_n$ has chance $\leq 15/1000000$ accepted by receiver.

Underlying fact: $\leq 15$ roots

$$m'(x) - m_1(x) - a' + a_1 \cdot m_1(x) - a' + a_1 + 10^6 \cdot m_1(x) - a' + a_1 - 10^6.$$ 

\textit{Warning:} very easy to break

simplified authenticator

$$- m_n[5] r^4 \mod p \mod 1000000:$$

$$m(x) - m_1(x) = a' - a_1.$$

Scaled up for serious security:

Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.

Assuming $\leq L$-byte messages:

Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$.

Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lceil L/16 \rceil / 2^{106}$.

e.g. $2^{64}$ forgeries, $L = 1536$:

$\Pr[\text{all rejected}] \geq 0.9999999998$. 

Authenticator is still secure for variable-length messages, if different messages are different polynomials mod $p$. 

No. Easy to prove: Every choice of $(n'; m'; a')$ with $m' \neq m_n$ has chance $\leq 15 = 1000000$ of being accepted by receiver.

Underlying fact: $\leq 15$ roots of $(m'(x))^m_1(a' + a_1 + 10^6) \cdot (a' + a_1 - 10^6)$.

Warning: very easy to break the oversimplified authenticator $(m_n[1] + \cdots + m_n[5] r^4 \mod p) + s n \mod 1000000:$$\text{solve } m'(x) = a' - a_1$. 

Scaled up for serious security: Poly1305 uses 128-bit $r$'s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.

Assuming $\leq L$-byte messages:
Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$.
Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lceil L/16 \rceil / 2^{106}$.

e.g. $2^{64}$ forgeries, $L = 1536$: $\Pr[\text{all rejected}] \geq 0.9999999998$. 

Authenticator is still secure for variable-length messages, if different messages are different polynomials mod $p$. 
Scaled up for serious security:
Poly1305 uses 128-bit $r$’s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.

Assuming $\leq L$-byte messages:
Each forgery succeeds for $\leq 8 \lceil L/16 \rceil$ choices of $r$.
Probability $\leq 8 \lceil L/16 \rceil / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lceil L/16 \rceil / 2^{106}$.

e.g. $2^{64}$ forgeries, $L = 1536$: $Pr[\text{all rejected}] \geq 0.9999999998$.  

Authenticator is still secure for variable-length messages, if different messages are different polynomials mod $p$.  

Authenticator is still secure for variable-length messages, if different messages are different polynomials mod $p$.  

Scaled up for serious security:
Poly1305 uses 128-bit \( r \)'s, with 22 bits cleared for speed. Adds \( s_n \mod 2^{128} \).

Assuming \( \leq L \)-byte messages:
Each forgery succeeds for \( \leq 8 \lceil L/16 \rceil \) choices of \( r \).
Probability \( \leq 8 \lceil L/16 \rceil /2^{106} \).

\( D \) forgeries are all rejected with probability \( \geq 1 - 8D \lceil L/16 \rceil /2^{106} \).

E.g. \( 2^{64} \) forgeries, \( L = 1536 \):
\( \Pr[\text{all rejected}] \geq 0.9999999998 \).

Authenticator is still secure for variable-length messages, if different messages are different polynomials mod \( p \).
Scaled up for serious security: Poly1305 uses 128-bit $r$’s, with 22 bits cleared for speed. Adds $s_n \mod 2^{128}$.

Assuming $\leq L$-byte messages: Each forgery succeeds for $\leq 8 \lfloor L/16 \rfloor$ choices of $r$. Probability $\leq 8 \lfloor L/16 \rfloor / 2^{106}$.

$D$ forgeries are all rejected with probability $\geq 1 - 8D \lfloor L/16 \rfloor / 2^{106}$.

e.g. $2^{64}$ forgeries, $L = 1536$: $\Pr[\text{all rejected}] \geq 0.9999999998$.

Authenticator is still secure for variable-length messages, if different messages are different polynomials mod $p$.

Split string into 16-byte chunks, maybe with smaller final chunk; append 1 to each chunk; view as little-endian integers in $\{1, 2, 3, \ldots, 2^{129}\}$. Multiply first chunk by $r$, add next chunk, multiply by $r$, etc., last chunk, multiply by $r$, mod $2^{130} - 5$, add $s_n \mod 2^{128}$.