Practical realisation and elimination of an ECC-related software bug attack

B. B. Brumley[1], M. Barbosa[2], D. Page[3], and F. Vercauteren[4]

[1] Department of Information and Computer Science,
Aalto University School of Science, P.O. Box 15400, FI-00076 Aalto, Finland.
billy.brumley@aalto.fi
[2] HASLab/INESC TEC
Universidade do Minho, Braga, Portugal.
mbb@di.uminho.pt
[3] Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK.
page@cs.bris.ac.uk
[4] Department of Electrical Engineering, Katholieke Universiteit Leuven,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium.
fvercaut@esat.kuleuven.ac.be

Abstract. We analyse and exploit implementation features in OpenSSL version 0.9.8*g* which permit an attack against ECDH-based functionality. The attack, although more general, can recover the entire (static) private key from an associated SSL server via 633 adaptive queries when the NIST curve P-256 is used. One can view it as a software-oriented analogue of the bug attack concept due to Biham et al. and, consequently, as the first bug attack to be successfully applied against a real-world system. In addition to the attack and a posteriori countermeasures, we show that formal verification, while rarely used at present, is a viable means of detecting the features which the attack hinges on. Based on the security implications of the attack and the extra justification posed by the possibility of intentionally incorrect implementations in collaborative software development, we conclude that applying and extending the coverage of formal verification to augment existing test strategies for OpenSSL-like software should be deemed a worthwhile, long-term challenge.

Keywords: elliptic curve, OpenSSL, NIST, fault attack, bug attack.

1 Introduction

Concrete implementation of cryptographic primitives is becoming easier as a result of more mature techniques and literature. Elliptic Curve Cryptography (ECC) is a case in point: twenty years ago ECC was limited to experts, but is now routinely taught in undergraduate engineering courses. However, such implementation tasks are still hugely challenging. This is because as well as functional correctness, the quality of an implementation is, in part, dictated by efficiency (e.g., execution speed and memory footprint) and physical security.

For (at least) two reasons, the efficiency of cryptographic primitives is an important issue within many applications. On one hand, many primitives represent an inherently expensive workload comprised of computationally-bound, highly numeric kernels. On the other hand, said primitives are often required in high-volume or high-throughput applications; examples include encryption of VPN traffic and full-disk encryption, both of which represent vital components in e-business. Both reasons are amplified because the primitive in question will often represent pure overhead at the application level. That is, cryptography is often an implicit enabling technology rather than an explicit feature: there is evidence to show it is common (and perhaps sane [11]) for users to disable security features in an application if it improves performance or responsiveness.

To summarise, some engineer must find an efficient way to map a complex, high-level specification of some primitive onto the characteristics of a demanding target platform, potentially using low-level programming languages and tools. Both the semantic gap between specification and implementation, and the skills gap between cryptography and engineering can be problematic. Two examples of the problems encountered, both relating to components in modern e-business work-flows, are as follows:

1. Nguyen [14] described an attack on GPG version 1.2.3, an open-source implementation of the OpenPGP standard. In short, the size of some security-critical parameters had been reduced; this meant computation was faster, but that the system as a whole was vulnerable to attack.
2. In part because of such wide-spread use, the open-source OpenSSL library has been subject to numerous attacks. Examples include issues relating to random number generation[5], and badly formulated control-flow logic allowing malformed signatures to be reported as valid[6].

Although other factors clearly contribute, one could argue that overly zealous optimisation is a central theme in both cases. Focusing on the provision of ECC in OpenSSL version 0.9.8*g*, this paper presents further evidence along similar lines. We stress that our aim is not to implicitly or explicitly devalue OpenSSL: one can, and *should*, read the paper more as a case study on the difficulty of cryptographic software implementation.

At the crux is an arithmetic bug, initially reported on the `openssl-dev` mailing list [16] in 2007 and later traced to the modular arithmetic underlying implementation of specific NIST elliptic curves; in short, the bug causes modular multiplications to (transiently) produce incorrect output. To the best of our knowledge, no cryptanalytic exploitation of this bug was previously known. Perhaps for this reason, it has not been considered a security risk, but rather a minor issue of functionality. Indeed, although the bug has been resolved in OpenSSL versions 0.9.8*h* and later it persists[7]; for example versions of the library are deployed in (at least) two major Linux distributions, namely Debian (as late as 5.0 "Lenny") and Ubuntu (as late as 9.10 "Karmic").

[5] `http://www.openssl.org/news/secadv_20071129.txt`
[6] `http://www.openssl.org/news/secadv_20090107.txt`
[7] `http://marc.info/?t=131401133400002`

The main contribution of this paper is a concrete attack: we show how the bug can be exploited to mount a full key recovery attack against implementations of Elliptic Curve Diffie-Hellman (ECDH) key agreement. The nature of the bug means the attack represents a software analogue (or a first practical realisation) of the bug attack concept [4] due to Biham et. al. Our attack works whenever the ECDH public key is static, and therefore reused across several key agreement protocol executions. In particular, any higher-level application relying on the SSL/TLS implementation of OpenSSL in the following two scenarios could be vulnerable:

1. Use of *static* ECDH-based cipher suites, (e.g., ECDH-ECDSA and ECDH-RSA). In such cipher suites, the TLS server holds a public key certificate that directly authenticates the ECDH public key; this is shared across an arbitrary number of key exchanges.
2. Use of *ephemeral* ECDH-based cipher suites (e.g., ECDHE-ECDSA and ECDHE-RSA) in combination with the OpenSSL *ephemeral-static* ECDH optimisation. In such cipher suites, and according to the TLS specification, a fresh ECDH public key should be generated for each key exchange. However OpenSSL allows one-time generation of said key when the TLS server is initialised, sharing it across an arbitrary number of key exchanges thereafter.

As a concrete example, we demonstrate the attack on stunnel version 4.42 (when linked against OpenSSL version 0.9.8*g*), an SSL-based tunnelling proxy.

As well as discussing potential countermeasures for vulnerable versions of the library, we also explore an alternative, longer-term solution. Specifically, we investigate use of formal verification as a means to *prevent* similar bugs rather than just detecting them a posteriori. This approach is particularly relevant in addressing the possibility of *intentionally* incorrect implementations, which could constitute a serious risk in software components developed using an open, collaborative approach. Our conclusion is that although such techniques can already play an important role, a step-change in attitude toward their use is required as software complexity increases; despite the effort required to adopt a development strategy that supports formal verification, this seems an important area for future work in the context of OpenSSL-like software.

From here on we use OpenSSL as a synonym for OpenSSL version 0.9.8*g* unless otherwise stated. In Section 2 we present a detailed analysis of features in OpenSSL that permit our attack to work, before explaining the attack itself, and possible countermeasures, in Section 3. In Section 4 we discuss approaches to formal verification that could help prevent similar defects in OpenSSL, and therefore similar attacks, and offer some concluding remarks in Section 5.

2 Background and analysis

The aim of this section is to relate high-level, standardised ECC with an analysis of associated low-level implementation features in OpenSSL which support our attack.

2.1 OpenSSL Implementation of NIST Standard Curves

For a GM-prime p, multi-precision integer multiplication modulo p can be particularly efficient. OpenSSL uses this fact to support ECC implementations over the NIST standard curves P-192, P-224, P-256, P-384 and P-521. Using P-256 as an example, we have

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

and, from here on, we refer to the resulting elliptic curve as E.

Assuming a processor with a 32-bit word size, imagine that given two 8-word operands $0 \leq x, y < p$, the goal is to compute $x \cdot y \pmod{p}$. Solinas demonstrates [17, Example 3, Page 20] that given $z = x \cdot y$, the 16-word integer product of x and y, one can compute $z \pmod{p}$ by first forming nine 8-word intermediate values

$$\begin{array}{ll}
S_0 = (& z_7,\ z_6,\ z_5,\ z_4,\ z_3,\ z_2,\ z_1,\ z_0\ ) \\
S_1 = (& z_{15}, z_{14}, z_{13}, z_{12}, z_{11},\ 0,\ 0,\ 0\ ) \\
S_2 = (& 0,\ z_{15}, z_{14}, z_{13}, z_{12},\ 0,\ 0,\ 0\ ) \\
S_3 = (& z_{15}, z_{14},\ 0,\ 0,\ 0,\ z_{10},\ z_9,\ z_8\ ) \\
S_4 = (& z_8,\ z_{13}, z_{15}, z_{14}, z_{13}, z_{11}, z_{10},\ z_9\ ) \\
S_5 = (& z_{10},\ z_8,\ 0,\ 0,\ 0,\ z_{13}, z_{12}, z_{11}\ ) \\
S_6 = (& z_{11},\ z_9,\ 0,\ 0,\ z_{15}, z_{14}, z_{13}, z_{12}\ ) \\
S_7 = (& z_{12},\ 0,\ z_{10},\ z_9,\ z_8,\ z_{15}, z_{14}, z_{13}\ ) \\
S_8 = (& z_{13},\ 0,\ z_{11}, z_{10},\ z_9,\ 0,\ z_{15}, z_{14}\ )
\end{array}$$

and then computing $S \pmod{p}$ with

$$S = S_0 + 2S_1 + 2S_2 + S_3 + S_4 - S_5 - S_6 - S_7 - S_8\,. \tag{1}$$

Note that $|S|$ cannot be much larger than p, meaning a small number of extra modular additions or subtractions, depending on the sign of S, would give the correct result.

OpenSSL adopts a similar approach for P-192, P-224 and P-521 but deviates for P-256 and P-384: we again use P-256 as an example, but note that the same problem exists for P-384. It proceeds using the following faulty algorithm: first it computes $t = S \bmod 2^{256}$ and the correct carry c (which is positive or negative) such that

$$S = t + c \cdot 2^{256}.$$

Note that per the comment above, the carry has a small magnitude; by inspection it is loosely bounded by $-4 \leq c \leq 6$, which is used from here on wlog. The result is computed, potentially incorrectly, via two steps:

1. set $r' = (t - c \cdot p) \bmod 2^{256}$, then
2. if $r' \geq p$, $r' = r' - p$.

The concrete implementation of these steps uses a fixed look-up table $T[i] = i \cdot p \bmod 2^{256}$ for small i, by computing $r' = t - \mathrm{sign}(c) \cdot T[|c|] \bmod 2^{256}$. The modular reduction in this case is implicit, realised by truncating the result to 8 words.

The intention is to eliminate any possibility of overflow; the assumption is that c is the exact quotient of division of S by p.

The reasoning behind the faulty algorithm is that if one writes $S = t + c \cdot 2^{256}$, then the exact quotient $q = S \div p$ is given by

1. if $c \geq 0$, then $q = c$ or $q = c + 1$,
2. if $c < 0$, then $q = c$ or $q = c - 1$

since c is small. Indeed, write $\Delta = 2^{256} - p$, then after subtracting $c \cdot p$ we obtain

$$S - c \cdot p = t + c \cdot 2^{256} - c \cdot p = t + c \cdot \Delta \,.$$

Since $-4 \leq c \leq 6$ and $\Delta < 2^{224}$, this shows the result is bounded by $-p < t + c \cdot \Delta < 2p$. The faulty algorithm therefore computes an incorrect result in the following cases:

- If $c \geq 0$, the algorithm fails when $t + c \cdot \Delta \geq 2^{256}$ since it computes r' only modulo 2^{256} and not as a full integer (for which the resulting algorithm would have been correct). Note that in this case the correct result would be $r' + \Delta$ and that modulo p, the correct result thus is $r' + 2^{256} \pmod{p}$.
- If $c < 0$, the algorithm fails when $t + c \cdot \Delta < 0$. The correct result then depends on whether $(t + c \cdot \Delta) \bmod 2^{256} \geq p$ or not: in the former case, the correct result is $r' - \Delta$, whereas in the latter case, the correct result is given by $r' + 2^{256} - 2\Delta$. Note that although there are two different subcases for $c < 0$, the errors $-\Delta$ and $2^{256} - 2\Delta$ are congruent modulo p, i.e. modulo p, the correct result is given by $r' - 2^{256} \pmod{p}$.

Note that Ciet and Joye [5, Section 3.2] consider the case of faults in the underlying field; the fault (resp. bug) here is certainly related, but occurs as a result of erroneous computation rather than corrupted parameters.

The resulting bug is difficult to detect using the (random) test vector approach employed by OpenSSL: it simply manifests itself too rarely. An upper bound for the probability of the bug being triggered can be obtained by ignoring the probabilities of certain carries occurring and analysing the case $t + 6 \cdot \Delta \geq 2^{256}$: if t was chosen uniformly over the interval $[0, 2^{256}[$, then this case occurs with probability less than 2^{-29}, so OpenSSL computes the result incorrectly with probability less than $10 \cdot 2^{-29}$.

To *deliberately* trigger the bug, we empirically arrived at the following strategies (after inspection of partial products within the integer product of x and y):

- For modular multiplication, selecting x, y as follows should induce an incorrect result for any random $0 \leq R_x, R_y < 2^{31}$:

$$\begin{aligned} x &= (2^{32} - 1) \cdot 2^{224} + 3 \cdot 2^{128} + R_x \\ y &= (2^{32} - 1) \cdot 2^{224} + 1 \cdot 2^{96} \ + R_y \end{aligned}$$

- For modular squaring, selecting $x = (2^{32} - 1) \cdot 2^{224} + 1 \cdot 2^{129} + R_x$, should induce an incorrect result for any random $0 \leq R_x < 2^{31}$.

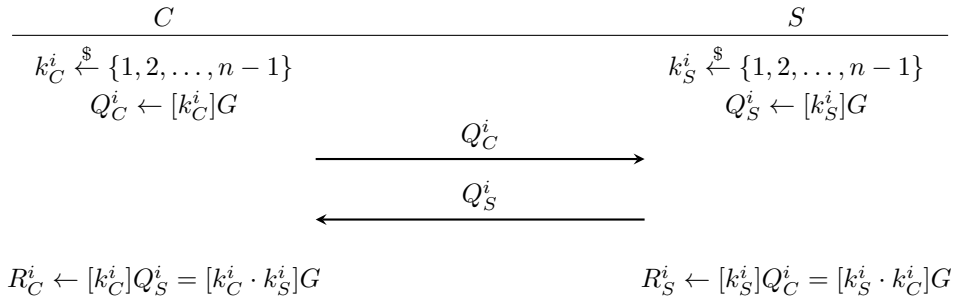

Fig. 1. A description of ECDH key exchange.

2.2 ECC Cipher Suites for TLS

Modern versions[8] of the Transport Layer Security (TLS) standard provide a number of different cipher suites that rely on ECC for key exchange. We focus on Elliptic Curve Diffie-Hellman (ECDH) and Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) based cipher suites[9]. To be precise, in these cipher suites the key exchange protocol is conducted to establish a secret key for a session i between a client C and a server S; it proceeds in three stages outlined in Figure 1, with client and server assumed to share $D = \{p, A, B, x_G, y_G, n, h\}$, a set of domain parameters. After the protocol terminates, $R_S^i = R_C^i$ represents the shared key.

Figure 2 illustrates the TLS handshake at a higher level of abstraction. We now describe how said handshake proceeds, detailing how the ECDH protocol messages formalised above are embedded in the communication. While our attacks are equally applicable in the case of client authentication, we omit the details for this case. The `ClientHello` message conveys the protocol version, supported cipher and compression methods, and a nonce to ensure freshness. The `ServerHello` message is analogous, but selects parameters from the methods proposed by the client (contingent on the server supporting them). The content of the `Certificate` message varies depending on the selected cipher suite:

- In ECDH-ECDSA, the `Certificate` message contains a static ECDH public key authenticated by a public key certificate signed with ECDSA; ECDH-RSA is analogous, but the public-key certificate is signed using an RSA signature. The static ECDH public key corresponds to the value Q_S^i above, and the server will reuse this value for multiple key exchanges with an arbitrary number of clients. For this reason, the `ServerKeyExchange` message is omitted in ECDH suites.
- In ECDHE-ECDSA, the `Certificate` message contains an ECDSA verification key, which is authenticated by a certificate signed with the same

[8] `http://tools.ietf.org/html/rfc5246`
[9] `http://tools.ietf.org/html/rfc4492`

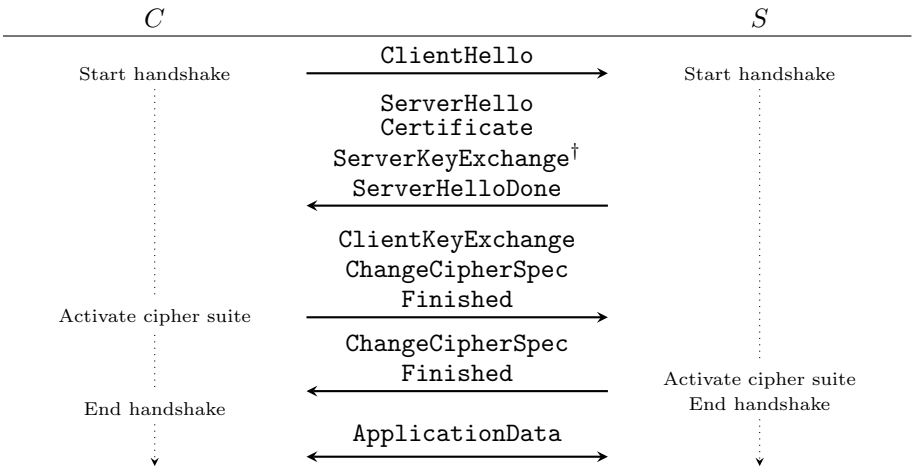

Fig. 2. Message flow in a TLS handshake with ECDH cipher suites; messages relating to client authentication are omitted for clarity, and those marked with † are only sent under specific circumstances.

algorithm; ECDHE-RSA is analogous but an RSA signature verification key is sent, and the public-key certificate is signed using an RSA signature. The server also sends message `ServerKeyExchange`, containing both a fresh ephemeral ECDHE public key (i.e., Q_S^i) and a digital signature authenticating this and other handshake parameters, including the exchanged nonces. Said signature is produced with either the ECDSA or RSA signing key matching to the verification key sent in the `Certificate` message.

The `ServerHelloDone` message marks the end of this stage of the protocol; the client then sends its ephemeral ECDHE key in the `ClientKeyExchange` message, which always includes a fresh ephemeral Q_C^i. Finally, the negotiated symmetric cipher suite is activated via the `ChangeCipherSpec` message. A session key for the cipher suite is derived from $R_S^i = R_C^i$ and the exchanged nonces. The `Finished` messages provide key confirmation for both parties, notably occurring client-first, and depend on all previous messages in the protocol execution.

2.3 OpenSSL Implementation of the ECC Cipher Suites

The ECDH implementation in OpenSSL is seemingly straightforward, and follows the TLS specification. However, the ECDHE implementation offers two distinct options for server applications. The first follows the specification and generates a new ephemeral ECDH key pair for every protocol execution. Deviating from

the specification, the second features an optimisation termed ephemeral-static ECDH[10].

When activated, the optimisation means a single ECDH key pair is generated during initialisation of the OpenSSL context within an application. This key pair is reused for all protocol executions thereafter; with ephemeral-static ECDH, OpenSSL has the server use a static key (i.e., a fixed k_i^S and hence Q_i^S for all i) for each OpenSSL context. Put another way, the key pair is ephemeral for each application instance and not (necessarily) per handshake instance. While this preserves forward secrecy between application instances, it violates forward secrecy within a single application instance when performing more than a single protocol execution. Interestingly, the default behaviour is the latter: to "opt out" and disable the optimisation, the application must explicitly use the `SSL_OP_SINGLE_ECDH_USE` option during initialisation of the context.

3 An Attack on ECDH in OpenSSL

Implementing scalar multiplication. For scalar multiplication on $E(\mathbb{F}_p)$, OpenSSL uses a textbook double-and-add algorithm along with the modified width-w NAF representation of k. For P-256 OpenSSL sets $w = 4$, i.e., each non-zero digit from digit set $\mathcal{D} = \{0, \pm 1, \pm 3, \pm 5, \pm 7\}$ is followed by at least three zero digits. Modified NAF is otherwise identical to traditional NAF but allows the most-significant digit to violate the non-adjacency property, if doing so does not increase the weight but reduces the length. This slight distinction between the two affects neither the derivation of our attack nor the implementation of it: henceforth we use NAF synonymously with traditional NAF.

Attack Goals and Limitations. The goal of the attacker is to recover the fixed k_S in the server-side computation of $R_S^i = [k_S]Q_C^i$. The algorithm we propose and implement recovers the digits of k_S by detecting faults in the server-side computation of R_S^i. The ability of the attacker to observe these faults heavily depends on the protocol and/or cryptosystem under attack. For example, when k_S is fixed but Q_C^i is not, it is uncommon for a protocol to directly reveal R_S^i to another participant. Inspecting TLS, one way the attacker can detect faults is by attempting to complete the handshake. If R_S^i is fault-free (denoted $R_S^i \in E$) then the computed session key is the same for both the client and server. Consider the phase of the handshake when the client sends `ChangeCipherSpec`, signalling that it has activated the newly negotiated session cipher suite, and transmits the encrypted `Finished` handshake message for key confirmation. Then, if the server successfully decrypts said message, the handshake continues, and ultimately succeeds. On the other hand, if R_S^i is faulty (denoted $R_S^i \notin E$) then the session keys differ, the server will not obtain the expected key confirmation message, and the handshake ultimately fails. The practical consequence is that the attacker *cannot* arbitrarily choose each Q_C^i in the protocol, rather he *must* know the discrete logarithm of said point in order to correctly calculate the negotiated session key.

[10] `http://tools.ietf.org/html/rfc5753`

The Attack Algorithm. Having established a method to detect the occurrence of faults, the algorithm proceeds in an exhaustive depth-first search for $\mathrm{NAF}(k_S)$ starting from the most-significant digit, trimming limbs and backtracking by iteratively observing handshake results. At each node in the search, the attacker submits a different carefully chosen point, termed a *distinguisher point*, to determine if the next unknown digit takes a specific value at the given iteration, tracing the execution path of the server-side scalar multiplication. We define a distinguisher point for an integer prefix a and target digit $b \in \mathcal{D} \setminus \{0\}$ to be a point $D_{a,b} = [l]G \in E$ such that $[a \parallel b \parallel d]D_{a,b} \notin E$ and $[a \parallel c \parallel d]D_{a,b} \in E$ for all $c \in \mathcal{D} \setminus \{0, b\}$ both hold. Here, l is known, a is the known portion of $\mathrm{NAF}(k_S)$, and d is any sufficiently long random padding string that completes the resulting concatenation to a valid NAF string. In practice, testing a single distinguisher point requires varying d over many values to ensure the computation reaches a sufficient number of possible subsequent algorithm states: this acts to deter false positives.

We step through the first few iterations to demonstrate the algorithm. For clarity, we use subscripts on variables a and $\mathcal{D}$ to identify the iteration, i.e., the digit index in the NAF representation from most- to least-significant, we are referring to. For $i = 1$, a_1 is the empty string and $\mathcal{D}_1 = \{1, 3, 5, 7\}$. The attacker finds a distinguisher point $D_{\emptyset,b}$ for each $b \in \mathcal{D}_1 \setminus \{1\}$ and uses these three points in attempted handshakes to the server[11]. Handshake failure reveals the correct digit, and allows us to set a for the next iteration as $a_2 = b$; if all handshakes succeed, the attacker deduces $a_2 = 1$ for the next iteration. Enforcing NAF rules, for $i = 5$ we have $a_5 = a_2 \parallel 000$ and $\mathcal{D}_5 = \{0, \pm 1, \pm 3, \pm 5, \pm 7\}$. The attacker then finds $D_{a_5,b}$ for each $b \in \mathcal{D}_5 \setminus \{0\}$ and uses these eight points in attempted handshakes to the server. Handshake failure reveals the correct $a_6 = a_5 \parallel b$ and if all handshakes succeed the attacker deduces $a_6 = a_5 \parallel 0$. The attack continues in this manner to recover all subsequent digits. On average, our attack takes 4 handshake attempts to recover subsequent non-zero digits, and 8 handshake attempts to detect zero digits (note that we do not explicitly check for zeros which are implied by the NAF representation).

Relation to the Bug Attacks of Biham et al. This algorithm relates to that which Biham et al. used to mount a bug attack against Pohlig-Hellman in the $\mathbb{F}_p^*$ setting [4, Section 4.1.1]. The authors consider recovering the binary digits of the exponent from a left-to-right binary modular exponentiation algorithm. It does this by finding input $X \in \mathbb{F}_p^*$ such that $(X^2)^2$ fails yet $(X^2)X$ does not, i.e., it uses the former to query explicitly for any zero digits and implicitly obtain any non-zero digits. Assume the attacker knows l such that $X = g^l$ (this is not necessary to carry out their attacks, but is necessary when adapting their strategy to attack TLS). The most-significant digit of the victim's binary string is a non-zero digit by definition. The attacker queries the next digit by submitting

[11] When $i = 1$ finding a distinguisher point $D_{\emptyset,1}$ is less practical as it can cause the table of pre-computed points to be erroneously populated, so in this case querying for that particular digit value occurs implicitly.

X. Assume wlog. that it fails: the attacker now knows the two most-significant digits are 1 and 0. To obtain the (or analogy of a) distinguisher point for the next iteration, the attacker simply computes $X^{1/2}$ and, knowing the discrete logarithm of X to the base g, can easily derive the logarithm of this group element. This procedure essentially cancels out the effect of the known digits, forcing the accumulator to take value X at the intended iteration.

In the discussion that follows we will see that, in our attack, we search for all distinguishing points independently. Indeed, although it is tempting to use existing distinguisher points to derive subsequent points, which would reduce the complexity of our attack, this approach does not seem to apply in our scenario. The distinguishing point derivation technique by Biham et al. works for $\mathbb{F}_p^*$ (and even $E(\mathbb{F}_p)$ when using affine coordinates) because there is a unique representation for group elements in both cases (i.e., elements of $\mathbb{F}_p$ are stored as their smallest non-negative residue). There are a number of reasons why this strategy is ineffective for the OpenSSL implementation of ECC, the most prominent being the use of projective coordinates: group element representations are not unique, as the implementation computes in an equivalence class. As a result, the attacker cannot force the accumulator to the desired value since it undergoes projective point doublings. To summarise, there is no obvious way to cancel out the effect of the known digits. In any case, we will see in the following that this does not impact on the practicality of our attack in a significant way.

Finding Distinguisher Points. Lacking an analytical method to derive distinguisher points, our implementation of the attack resorts to random search. This is done by choosing l at random and testing whether $D_{a,b}$ satisfies the properties of a distinguisher point and, as previously mentioned, varying d over a reasonable amount of random values. The practicality of carrying out the attack thus hinges (in part) on the effort required to find said points, i.e., the average number of l values to test. Figure 3 (left) depicts our observed effort in two different parts of the attack. The solid line represents effort required at the beginning of the attack with less than 12 digits recovered. The dashed line represents effort required towards the end of the attack with roughly 224 digits recovered. This empirical data suggests not only that the computational effort to find a specific distinguisher point is rather modest, but that said effort does not significantly depend on the amount of known key digits. That is, as the attacker recovers successive digits, the effort to find new distinguisher points remains fairly constant. It is worth mentioning that the search scales perfectly with the number of computing nodes.

Attack Analysis and Results. The practicality of the attack additionally hinges on the number of required distinguisher points, i.e., the average number of attempted handshakes to recover the entire key. Our theoretical analysis of the expected number of queries, based on a rough approximation to the distribution of digits in the NAF representation of a random 256-bit secret exponent, points to an approximate value of 635 handshakes, suggesting that said value is similarly

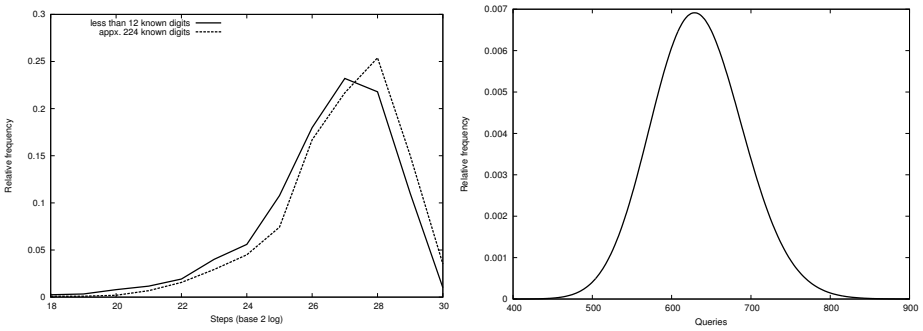

Fig. 3. Left: distribution of required search steps to find distinguisher points (lg-linear). The solid line represents effort towards the beginning of the attack (mean 26.5 s.d. 2.0) and the dashed line towards the end (mean 26.9 s.d. 1.9). Right: distribution of required queries to the server, or distinguisher points, for the full attack (mean 633.2 s.d. 57.7).

modest. We also measured this value empirically and obtained a consistent result, as illustrated in Figure 3 (right).

The proof-of-concept source code for our attack implementation includes distinguisher points for all NAF strings up to length 12. As it stands this immediately removes roughly 12 bits of entropy from the private key, and is of course easily extendible. The code includes instructions for running the attack in two different use cases:

1. The stunnel application provides a flexible SSL proxy for any application that does not natively support SSL; it links against OpenSSL to provide the SSL functionality. When stunnel is configured to support ECDH suites with a static ECDH key, our attack implementation facilitates recovery of said private key. Once carried out, it allows the attacker to decrypt all previous SSL sessions and to impersonate the server indefinitely.
2. The s_server application within OpenSSL is a generic SSL server. This application, and those similar to it supporting ECDHE suites, are vulnerable since they feature the ephemeral-static ECDH optimisation. The attack implementation facilitates recovery of the application instance's ECDH private key. Once carried out, it allows the attacker to decrypt all previous SSL sessions from the application instance and to impersonate the server until the application restarts.

Algebraic and Algorithmic Countermeasures. Coron outlines three methods to thwart DPA attacks [7, Section 5]. In general, they seek to counteract the deterministic nature of double-and-add style scalar multiplication routines, and are therefore quite effective against the bug attack presented above.

Scalar blinding, i.e., $[k]P = [k + rn]P$ for (small) random value r, effectively randomises the execution path of the scalar multiplication algorithm. This is not "free" however: the performance overhead (and security) is proportional to the size of the random multiplier. Point blinding, i.e., $[k]P = [k](P + R) - S$, with

randomly chosen $R \in E$ and $S = [k]R$ (updating both $R = [r]R$ and $S = [r]S$ for small, random r periodically), is equally effective. However, this also entails some performance overhead and is slightly more intrusive to integrate. Lastly, coordinate blinding, i.e., multiplying the projective coordinates of the accumulator by a random element of the finite field in such a way that preserves the equivalence class, effectively randomises the states of the scalar multiplication algorithm. In this case, said blinding would only need to occur at the beginning of the algorithm and hence does not entail anywhere near as significant a performance overhead. Our implementation as a patch to the OpenSSL source code is available from the `openssl-dev` mailinglist[12].

Algorithmic countermeasures seem ineffective against the attack if they are deterministic. The Montgomery ladder [13, Section 10.3.1], for example, can resist many forms of side-channel attack due to the regular nature of operations performed; it cannot resist a varient of the attack in Section 3 however, since one can still select distinguisher points that target the control-flow and hence (iteratively) recover the scalar. See the full version of this paper for a more detailed discussion.

4 Approaches to Formal Verification

In this section we investigate whether it is realistic to use current formal verification technology to prevent similar bug attacks in open-source projects such as OpenSSL. We focus our analysis in two complementary aspects of this problem: first the design of efficient algorithms for carrying out the necessary numeric computations, and second checking that these algorithms are correctly implemented in machine code. The concrete arithmetic bug that we explore in this paper serves as a perfect illustration of why these two aspects should be considered separately.

A high-level specification of the procedure for modular reduction that was found to be incorrect is described in Section 2.1. Producing a concrete implementation of this procedure implies a need to refine it into a lower-level specification; the particular refinement we are analysing can be described as follows:

1. Pre-compute a table T, where the i-th element $T[i] = i \cdot p$ (for small i).
2. To reduce the integer product $z = x \cdot y$ modulo p, first construct the intermediate values $S_0, S_1, \ldots, S_8$ based on the representation of z.
3. Write the integer sum $S = S_0 + 2S_1 + 2S_2 + S_3 + S_4 - S_5 - S_6 - S_7 - S_8$ as $S = t + 2^{256} \cdot c$.
4. Return the result $r' = t - \mathrm{sign}(c) \cdot T[|c|] \pmod{2^{256}}$.

This highlights a subtle point: rather than a programming error, the bug is more accurately characterised as a design error. That is, the *incorrectly designed* refinement is *correctly implemented* in OpenSSL.

We next discuss how one can formally verify the design of such refinements using existing technology. We discuss the viability of fully verifying implementation correctness in the full version of this paper.

[12] `http://marc.info/?l=openssl-dev&m=131194808413635`

The first question we consider is whether it is feasible to verify that a particular refinement is correct wrt. the associated high-level specification. In order to illustrate the techniques that can be employed at this level, we present two examples inspired in the bug described in the previous sections; each represents a refinement (invalid and valid respectively) along the lines above.

We have used the CAO domain specific language for cryptography [3] and the CAOVerif deductive verification tool [19]. The CAO language is syntactically close to C, but is equipped with type system (including, in particular, multi-precision integers) that make it straightforward to express mathematically-rich algorithms. The CAOVerif tool [13] takes an annotated version of the program one wishes to prove correct as input (the specification of correctness is included in the annotations), and generates a set of proof obligations that need to be validated as output. If one is able to validate all the proof obligations, this implies the program meets the specification. The proof obligations can be discharged by automatic provers such as Simplify [8] or Alt-Ergo [6] or, if these fail, one can use an interactive theorem prover such as Coq [18]. The verification condition generation procedure is based on Hoare logic [12], and uses the Jessie/Frama-C [9] and Why [10] platforms as a back-end.

Failed Proof for an Incorrect Refinement. Proving that the refinement used by OpenSSL is functionally equivalent to the original specification essentially reduces to proving that steps 1, 3 and 4 compute the same result as Equation 1. To illustrate how the proof proceeds we implemented these steps in CAO, annotating the result using the CAO Specification Language [2] (closely inspired by the ANSI C Specification Language) to indicate the proof goal. The most relevant fragment is presented below, where `Prime` and `Power2` are global variables holding the values of p and 2^{256}, respectively:

```
1  typedef modPower2 := mod[2**256];
2
3  /*@ requires ((0 <= sum) && (sum <= 7*(Power2-1))
4      && (Prime == 2**256 - 2**224 + 2**192 + 2**96 - 1)
5      && (Power2 == 2**256))
6      ensures  ((0 <= result) && (result < Prime)
7                && (exists d : int; result + d*Prime == sum)) */
8  def modPrime(sum : int) : int {
9      def c,res : int;
10     c := sum / Power2;
11     /*@ assert c >= 0 && c <= 6 */
12     /*@ assert 0 <= (sum - c*Prime) */
13     /*@ assert (sum - c*Prime) < Power2 */
14     res := (int)((modPower2)sum - (modPower2)(c*Prime));
15     if (res >= Prime) {
16        res := res - Prime;
17     }
18     return res;
19 }
```

Ignoring the embedded annotations for the moment, `modPrime` takes the summation `sum` as input (which for simplicity and wlog. we assume to be positive),

[13] A distribution of the CAOVerif tool and source code for the examples in this paper are available from `http://crypto.di.uminho.pt/CACE/`.

and computes a (possibly incorrect) output of `sum` modulo `Prime`. This computation is performed in three steps that mimic the OpenSSL implementation: 1) it calculates the (computationally inexpensive) division by `Power2`, 2) it uses the result `c` to subtract a multiple of `Prime` from the input (this operation is carried out efficiently modulo `Power2` by casting the values to an appropriate data type), and 3) the result is placed in the correct range by applying a conditional subtraction.

The annotations in the code can now be described. The specification of `modPrime` is a contract including a precondition `requires` and a post-condition `ensures`. It states that provided `sum` is in the correct range, i.e., $0 \leq$ `sum` $\leq 7 \cdot$(`Power2`-1), and that the output meets the mathematical definition of the least residue of `sum` modulo Prime, i.e., $(0 \leq$ `res` $<$ `Prime`$) \wedge (\exists d$ st. `res` $+ d \cdot$ `Prime` $=$ `sum`$)$. Inside the function, a series of assertions guide the proof tool toward establishing intermediate results towards an attempted proof. For example, one is able to establish the correct range of `c` after the division. The proof fails, however, when one tries to establish that performing the subsequent calculation modulo `Power2` will produce a result that is still congruent with `sum` modulo `Prime`. In particular, one will not be able to prove (if that was the initial intuition) that `sum` $-$ `c` $\cdot$ `Prime` $<$ `Power2` which would be sufficient to ensure that the calculations *could* be performed modulo `Power2`.

Robust Proof for a Correct Refinement. Consider the following alternative refinement to that presented above.

```
1  /*@ requires ((0 <= sum) &&& (sum <= 7*(Power2-1))
2      &&& (Prime == 2**256 - 2**224 + 2**192 + 2**96 - 1)
3      &&& (Power2 == 2**256))
4      ensures  ((0 <= result) &&& (result < Prime)
5                &&& (exists d : int; result + d*Prime == sum)) */
6  def modPrime(sum : int) : int {
7          def res : int := sum;
8      /*@ ghost def t : int :=0; */
9      /*@ assert res + t*Prime == sum */
10     /*@ invariant (res>=0) &&& (res == sum + t*Prime) */
11     while (res>=Prime) {
12         /*@ ghost t:=t-1; */
13         res := res - Prime;
14     }
15     /*@ assert ((0 <= res) &&& (res < Prime)
16                              &&& (res + (-t)*Prime == sum)) */
17     return res;
18 }
```

This implements the "natural" refinement: it simply subtracts the `Prime` from the input until the result is in the appropriate range. In order to complete the proof, one needs to include a loop invariant that keeps track of how many times `Prime` is subtracted; to achieve this, we use a "ghost" variable `t` that is only visible to the verification tool. The annotated result can be fed to the CAOVerif tool, which will automatically check that the program indeed meets the specification (noting that this automation relies partly on the assertions included).

5 Conclusions

This paper presents a concrete attack against ECDH-based functionality supported by OpenSSL version 0.9.8*g*. The attack works whenever the ECDH public key is static: this may occur either explicitly as a result of the selected cipher suite, or (partly) implicitly as a result of the (non-standard) ephemeral-static optimisation supported by OpenSSL. It is worth noting that we also considered exploiting the bug to mount invalid curve attacks [1]: while this allowed us to bypass OpenSSL point validation routines, it did not lead to a practical attack due largely to the nature of the bug severely limiting the number of invalid curves.

The arithmetic bug has been resolved in OpenSSL versions 0.9.8*h* and later. As a result, it is tempting to conclude that the attack does not represent a serious threat. However, vulnerable versions of the library are deployed in (at least) two major Linux distributions, namely Debian (as late as 5.0 "Lenny") and for Ubuntu (as late as 9.10 "Karmic"). Although they selectively apply patches to the default installation, the arithmetic bug persists in both. That is, although a patch resolving the bug has been available since 2008, it has yet to permeate existing installations. This represents a concrete example of the premise that patching is no panacea for similar problems.

Whether OpenSSL should prevent optimisations like ephemeral-static being included or invoked is perhaps a more philosophical question aligned to a bigger picture. For example, problems relating to IPsec support for encryption-only modes of the Encapsulating Security Payload (ESP) protocol seems conceptually similar; a comprehensive overview and resulting attack is given by Paterson and Yau [15]. One can conjecture that the motivation for encryption-only ESP, like ephemeral-static ECDH, is efficiency. Given that provision of a more efficient, less secure option will inevitably lead to someone using it, our work lends weight to the argument that a better approach may be to permit *only* secure, albeit less efficient options.

While (non-)support for various options in OpenSSL is subjective in part, the correctness of what *is* supported is less debatable. As such, detecting bugs in a more rigorous manner represents a difficult and extremely resource- and time-consuming task if undertaken over the entire implementation. OpenSSL clearly differs from a formal cryptographic standard, but it represents a de facto, ubiquitous and mission-critical software component in many settings. As such, we suggest that the effort required to adopt a development strategy capable of supporting formal verification is both warranted, and an increasingly important area for future work.

Acknowledgements. This work has been supported in part by EPSRC via grant EP/H001689/1 and by project SMART, funded by ENIAC Joint Undertaking (GA 120224). This work is part-financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project ENIAC/2224/2009 and by ENIAC Joint Undertaking under grant agreement number 120224.

References

1. A. Antipa, D.R.L. Brown, A. Menezes, R. Struik, and S.A. Vanstone. Validation of elliptic curve public keys. In *Public Key Cryptography (PKC)*, pages 211–223. Springer-Verlag LNCS 2567, 2003.
2. M. Barbosa. CACE Deliverable D5.2: formal specification language definitions and security policy extensions, 2009. Available from `http://www.cace-project.eu`.
3. M. Barbosa, A. Moss, and D. Page. Constructive and destructive use of compilers in elliptic curve cryptography. *J. Cryptology*, 22(2):259–281, 2009.
4. E. Biham, Y. Carmeli, and A. Shamir. Bug attacks. In *Advances in Cryptology (CRYPTO)*, pages 221–240. Springer-Verlag LNCS 5157, 2008.
5. M. Ciet and M. Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, Codes and Cryptography*, 36(1):33–43, 2005.
6. S. Conchon, E. Contejean, and J. Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006. `http://ergo.lri.fr/papers/ergo.ps`.
7. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 292–302. Springer-Verlag LNCS 1717, 1999.
8. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
9. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 15–29. Springer-Verlag LNCS 3308, 2004.
10. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, pages 173–177. Springer-Verlag LNCS 4590, 2007.
11. C. Herley. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *New Security Paradigms Workshop (NSPW)*, pages 133–144, 2009.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
13. P.L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.
14. P.Q. Nguyen. Can we trust cryptographic software? Cryptographic flaws in GNU Privacy Guard v1.2.3. In *Advances in Cryptology (EUROCRYPT)*, pages 555–570. Springer-Verlag LNCS 3027, 2004.
15. K.G. Paterson and A.K.L. Yau. Cryptography in theory and practice: The case of encryption in IPsec. In *Advances in Cryptology (EUROCRYPT)*, pages 12–29. Springer-Verlag LNCS 4004, 2006.
16. H. Reimann. BN_nist_mod_384 gives wrong answers. openssl-dev mailing list #1593, 2007. Available from `http://marc.info/?t=119271238800004`.
17. J.A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research (CACR), University of Waterloo, 1999. Available from `http://www.cacr.math.uwaterloo.ca/techreports/1999/corr99-39.pdf`.
18. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. `http://coq.inria.fr`.
19. B. Vieira, M. Barbosa, J. Sousa Pinto, and J.-C. Filliatre. A deductive verification platform for cryptographic software. In *International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert)*, 2010.