Master research Internship

Bibliographic report

Dolev-Yao-Model-Guided Fuzzing of E-Voting Protocols

Domain: Cryptography and Security

Author:
Benjamin Voisin

Supervisor:
Lucca Hirschi
Alexandre Debant
PESTO - Loria/Inria

Abstract

Electronic-voting is the process of using online devices such as computer or smartphone to participate in an election. Despite being critical software, electronic-voting systems have been found to contain usable vulnerabilities. Today, many tools exist to mathematically model a voting protocol, and to use this models to prove some security properties. However, the mathematical proofs cannot say anything about the software implementations of the existing protocols, and there might be an important gap between what the protocol is supposed to do, and the comportment of the underlying software.

Software testing methods exists, but they rarely exploit the specific aspects of an e-voting protocol, and the specific needs in terms of security. Dolev-Yao-Model-Guided fuzzing is an approach to use the mathematical models used for formal proofs to provides better testing for protocol implementations. However, DY-fuzzing has yet to be used for e-voting protocol, as they raise many more challenges than the protocols previously fuzzed.

Contents

1 Introduction

Electronic-voting is a more and more widely used tool. It gives the possibility to make large-scale election for a fraction the cost, to expand accessibility by giving people the ability to vote from home. Many elections today use e-voting, even for important political election. It is used for political elections in Estonia [17], Switzerland [13] and France [15].

However, the security of such systems is not straightforward. In the past, some actually used have been shown to be broken ([15, 9])

This bibliographic report aims to present how e-voting protocols are secured today. First, by describing in Section 2 e-voting protocols and what are the desired properties for a secure e-voting protocol. Then, in Section 3 how to formally describe a protocol and its security properties to make sure that it is safe to use.

Secondly, we discuss securing the actual implementation of a cryptographic protocol, by discussing the *fuzzing* method in Section 4, and then expanding this technique to *Dolev-Yao-Model-Guided fuzzing* (DY-fuzzing) in Section 5, a technique combining the formal framework from Section 3 with the fuzzing approach from Section 4 in order to obtain better results specifically for security protocols.

Finally, in Section 6, we discuss the main goal of this internship and why it is not straightforward : Making DY-fuzzing work with electronic-voting.

Electronic-voting is the process of setting an election where voters can cast their vote from their own device connected to the internet, like a computer or a smartphone. In this report we do not look at the use of voting machines used to assist physical voting. The use of electronic voting makes large-scale attacks possible much more easily than with physical voting. So, to achieve similar level of security, electronic-voting systems must be properly secured and verified.

Formal verification is the process of modelling a protocol in a formal framework, and using this to mathematically prove security properties. This can be done by hand, or with proving assistants like [6, 19, 2, 5]. This is done very often for electronic systems, either by the designer to prove that it is indeed secure, or by academics to find protocol vulnerabilities before they can be used. However, formal verification only tackles the *protocol* part, and is completely independent of the actual software. This means that even formally proved protocol can still host critical vulnerabilities in software bugs.

Fuzzing is a method for finding such implementation bugs. The core idea is to repeatedly runs the program that needs to be tested against varying inpt. By running it like that millions of times, possible bugs are found faster. Of course, this does not provide any guarantee and cannot find every bug of a software, but the fuzzing approach has proven to be very effective and is an automated step for most major and security-critical software programs.

DY-fuzzig is an attempt to improves fuzzing in the context of cryptographic protocols. It uses the methods for formal verification in order to produce more complex fuzzing inputs than random bytestrings. This allows DY-fuzzing to reach deeper states of the tested software and to find logical, protocol-related vulnerabilities instead of just crashes and memory bugs.

However, DY-fuzzing has currently only used for simple client-server protocol like TLS or SSH. E-voting protocols are much more complex in the number of parties involved, in the security properties that need to be verified. Additionally, e-voting protocols have varying *trust models*, meaning that the protocol must be able to run properly even if, for example, a voting server is malicious. This kind of inside attack is not yet possible to express in DY-fuzzing.

2 Electronic voting

E-voting is an increasingly used tool to improve the accessibility of elections and reduce their costs. However, the organization of such elections can be of critical importance and subject to bad actor willing to change the result. This makes e-voting software tools that need to be strongly secured and resilient against bad actors.

Some countries have legal requirements for e-voting systems ([8] in France or [7] in Switzerland). These requirements need to be both properly defined and check before using any e-voting system.

This section explains the different properties that are needed in a secure e-voting protocol, with the secrecy in Section 2.1 and verifiability in Section 2.2, and finally describes the working of the deployed and used SwissPost protocol in Section 2.3. The formalization of the properties discussed here are detailed in Section 3.2.

A voting system typically has 4 agents that needs to be taken into consideration : the voters, the voting server, the decryption authorities, and eventual auditors. The voting server host the election and receives the votes cast by voters. The decryption authorities perform the final tally with the data from the voting server. And finally auditors watch the going of the election to try to detect malicious activities. Of course, in addition to that, we need to take into account possible *attacker*, malicious actors that will try to break the system.

2.1 Vote secrecy

The first important property of e-voting is vote secrecy. It must be impossible to know which vote was cast by someone.

In physical voting, this property is achieved thanks to two mechanisms : votes are secretly put in an envelope, and once inside the ballot box, the envelopes are mixed before the tally to make it completely impossible to link a ballot to one voter. These two mechanisms have equivalents in e-voting (encryption of ballots and the use of mixing networks [21, Section 4.5], and the use of homomorphic encryption to avoid having to decipher individual ballots).

The secrecy of the vote is not only necessary to protect voters from unhappy opponents, it's also critical to guarantee that one's vote cannot be bought or forced with some form of coercion. If we can learn ones vote, we can buy it, or put a gun to the voter's head in order to force him to vote for something. Voters might also be subject to peer-pressure, and would not be able to freely express their true will. The inability for a voting system to support such influences is called *coercion resistance*.

Between the 15 and the 17th of January 1793, the deputies of the Convention national voted to determine the fate of king Louis XVI. It is still possible today to know who voted for what. However, these choices revealed to be of crucial importance. In the years following, people who did not choose to punish Louis XVI faced severe consequences [11]. A vote at some point, even if it may seem insignificant, can be of great importance in the future if its secrecy is not guaranteed. This is the *everlasting privacy* property.

Cryptographically, everlasting privacy is hard to guaranty. An encrypted ballot now might be decrypted later. The increase of computing power, the finding of possible flaws in the encryption scheme, and the arrival of quantum computers could all help break an encrypted ballot.

One way to solve this issue, is to not save the voter's identity with the ballot. Thus, even if the ballot encryption is broken, we cannot link the vote cast to one's identity. However, this makes it impossible to verify that all ballots do really correspond to ballot cast by legitimate voters (see Section 2.2). A ZKP (Zero Knowledge Proof) must then be used to have both property at the same time, as done in [14].

2.2 Verifiability

Another crucial aspect for e-voting is Verifiability. A voter, or anyone auditing an election should be able to check some properties about the voting process. There are multiple verifiability properties that can be wanted in an e-voting protocol, each providing a certain level of confidence over the ongoing election.

Verifiability properties can be split in 4 categories :

- Cast-as-Intended
- Recorded-as-Cast
- Universal verifiability
- Eligibility

The first two concern one voter that want to checks that his ballot is properly accounted, whereas the other twos concerns auditors who want to checks that there was no mangling in the voting process (of course any voters might want to audit the process of the election).

The combination of all these properties result in an *end-to-end* verifiable voting protocol, meaning that between the intention of the voters and the results, there has been no modification from the servers, the authorities detaining decryption keys, or any entity lawfully or not part of the voting system.

With every verifiability property comes *accountability*, mainly, if a verification fails, whose fault it is ? This means that in addition to verification, e-voting protocol must incorporate accountability elements, and plan beforehand for *dispute resolution*.

Recorded-as-Cast is from the individual view of one voter. The voter must be able to check that his ballot is correctly registered in the ballot box. In e-voting, this often mean that an action must be executed after the vote is cast, as the distance between the voter and the server collecting the votes makes it impossible *a priori* to be certain that a ballot sent will be registered and still be present at the end of the election.

Cast-As-Intended is linked to individual verifiability, and is also done individually by the voter, but here he wants to checks that his vote present in the ballot box contains his real vote. It aims to protect against a possible malware on the voting device that would modify the ballot before it reaches the ballot box.

This is one of the most difficult properties to guarantee for a voting system, especially while preserving coercion resistance. Intuitively, it seems that if you can prove what you voted for, then your vote can be bought or forced upon you. One way to circumvent this is not to try to prove what is inside the actual vote, but to check the device casting the vote cannot modify the vote without the voter noticing.

Multiple propositions exist to provide Cast-as-Intended. The Estonian protocol [17] and [22] make use of a second device to check that the encrypted ballot is correct, by giving the randomness used for the encryption. The Benaloh challenge [4] is similar, but the voter choose if the vote is a real vote, or an audit, before the vote, so that the audited ballots are not the same as the real ones. The SwissPost protocol [13] (see Section 2.3) uses return codes sent by postal mail, ones for each possible choice. Once the vote is cast, the voting device shows the corresponding code, and the voter can check that it's the same as the one received via postal mail.

Universal verifiability is about the next phase of the election, when the results are computed. Anyone must be able to check that the result really is the sum of all the votes that are present in the ballot box. This often involves a mathematical proof that the result is really the sum of all the ballot present in the ballot box.

Eligibility is intended to protect against the addition of unlawful ballots, for example from the voting server. Anyone auditing the election must be able to check that each of the ballots present in the ballot box is associated to a legitimate and eligible voter.

To be able to check the eligibility of a ballot, one can publish a public list of credentials with associated private keys. Each voter receives a private key to sign the cast ballots, as done in Belenios [12]. However, in this configuration, one party may know the link between the public credentials and the real identity of the voters. It is possible to use ZKP to solve this issue, by making transferable and anonymous proof of legitimacy, as in [14].

End-to-End Verifiability If any voter can verify that the ballot they cast contains their original voting intention (Cast-as-Intended), that this ballot is properly recorded by the ballot box (Recorded-as-Cast), and that any other votes in the ballot box are from legitimate voters (Eligibility) and properly accounted in the tally (Universal verifiability), then you have a proper End-to-end verifiable voting protocol.

2.3 SwissPost

In this section, we will discuss in more details the SwissPost protocol. Because it is one of the few actually open-source and used in political elections, it is a good candidate for our fuzzing tests.

SwissPost is a protocol developed by the SwissPost company (the postal company of Switzerland) to serve for political elections in Switzerland. It was introduced in 2023 for political voting in the Swiss cantons. One very interesting element of SwissPost, is that its security properties are written into the Swiss law, and the Swiss law enforce the *Cast-as-Intended* property.

This means that even if the voting device is corrupted, the voter should be able to detect if their vote has been modified. This is achieved with *Return Codes*, sent to each voter via postal mail. An example of a voting car can be found in Figure 1, it contains all the information necessary to carry the vote :

- The Login code, to authenticate and cast the vote.
- The Return codes that the voter need to check against what is shown by the voting device.
- The Approve code, that the voter enters only if the Return code matches.
- The Finalization code that is shown to the voter at the end of the voting phase to prove that the ballot has been registered.

The flow of the vote is shown in Figure 2. The CC_j at the right of the figure are the **Control Components**. In practice, there are 4 of them so they can perform *secret sharing* : they each have partial codes for the Return and Finalization codes, that are xored by the voting device. This means that if at least one of them is honest, then the other Control Components cannot learn the actual return code and so break the privacy of the vote.

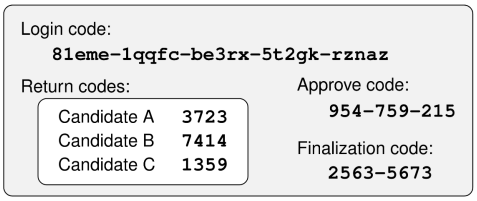

Figure 1: A Voting Card received by a Swiss voter (from [13])

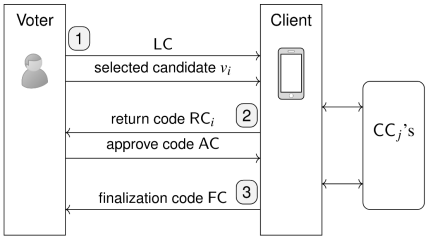

Figure 2: Voter's view of the voting phase. Step 1: authentication and candidate selection; Step 2: check validity and confirmation; Step 3: final check. (from [13])

The final part of the SwissPost voting protocol is the **Setup Component**. It's this part of the protocol that is responsible for generating the voting material, giving it to SwissPost (the postal company), and giving the partial codes to the Control Components. In the version that we intend to fuzz, this component needs to be trustworthy, and thus is a critical part of the system. Efforts have been made in [13] to strengthen the threat model, of the setup component, but is not yet implemented in code that can be tested.

In the past, vulnerabilities have been found on the SwissPost protocol [9] and its predecessor CHVote [10], and SwissPost itself is requesting for researcher to try to find bugs both in the protocol and in its implementation, as it started a bug-bounty program [23].

3 Formal proof of security

Once we know what properties we want our voting protocol to achieve, it is of paramount importance to ensure that it really satisfy the properties we want.

One way to achieve that is with *formal proofs*, namely mathematical proofs that guarantee the properties of our voting protocol. For example, we might prove that the individual verification always implies that the vote is properly cast to the ballot box. It is also possible to prove that it's impossible to know what someone decided to vote.

Such proofs often consists of modeling an attacker, what capabilities it has, and then proving that with these capabilities it's impossible to break the security properties.

3.1 The Dolev-Yao model

There exist many ways of modelling a protocol. One solution is symbolic models, firstly introduced by Dolev and Yao [16]. The idea is to use first-order terms to represent cryptographic message, and an equational theory to represents what is possible to do with the different cryptographic functions. This is called the *symbolic model*.

In this model, messages are described using a term algebra. We define a set of function symbols with their arity that are needed to model the protocol (e.g., `aenc/2`,`sdec/2`), and, to give a semantics to the function symbols, an equational theory over the function symbols. For example, if our protocol needs symmetric encryption, we would need the functions `senc/2` and `sdec/2`, and the equation $\texttt{sdec}(\texttt{senc}(x, key), key) = x$, meaning that the symmetric decryption with key of x encrypted with key gives back x.

In this model, a protocol is made of multiple *process*, algorithm of the protocol that will interact over one or multiple *channels*. A process is simply a succession of *in* and *out*, it can listen to messages over the network (*in*) and emits messages on the network (*out*), with the

addition of some conditional logic, and the ability to apply function symbols to the received terms.

For example, the process `in(c,x); out(c, aenc(x, pk(y)))`. is a process that listen for a message, and then outputs the same message, encrypted with the public key of `y`. Here, `c` is a *channel*, a place where message can be sent. In some protocols it can be useful to have multiple channels for different messages.

The presence of this process means that, in addition to the equational theory of the function symbols for the protocol, an attacker sends a message to the process, and get back the message encrypted with the public key of y.

An important part of the DY-model is the power of the attacker. It is assumed that the attacker has complete control over every channel. This means that it can intercept, modify, delete and add new messages. When proving things within this model, we check what terms are derivable from the initial knowledge of the attacker, the equational theory, and the different process running. For example, the last process, if repeated, means that for every term x that the attacker knows, it can deduce the term $\texttt{aenc}(x, \texttt{pk}(y))$.

A *trace* is a sequence of action, inputs or outputs, and their contents. It corresponds to one execution of the protocol. It incorporates both actions from honest players and the actions done by the attacker.

It is important to note that the symbolic model is not perfect, and things proven in the symbolic model might still have vulnerabilities. In the symbolic model, we model the computation that an attacker can do, whereas the cryptographic primitives state what problem is *difficult* for an attacker (meaning impossible to solve in polynomial time).

To solve this, some work has been done to develop the *computational model*, where the attacker is a polynomial time Turing machine. See for example Squirrel [2] and Cryptoverif [5], but due to the added complexity of the model, proofs in the computational model often targets smaller parts of a protocol, and cannot be automated as easily as with the symbolic model.

3.2 Properties of security

We now need to express the security properties that we want to prove for our protocol. In the DY-model, there are two main types of security properties : *trace-based* properties, that indicates that nothing bad happened, and *equivalence-based* properties, that establish that an attacker is not able to distinguish between two different scenarios.

For stating trace-based properties, we need to add information within the process, named *events*. An event can record the value of a term at a specific moment of the process. For example, to represent the correct end of a verification process for a given voter and vote cast, we can put an event `HappyVoter(id,vote)` at the end of the process.

With an event on the voting-server process, `BallotRegistred(ballot)` after receiving and registering a ballot, we can write the property :

```
event(HappyVoter(id, vote)) =>
    BallotRegistred(ballot) & ballot = aenc(pk, vote, random)
```

This property means that, every time a voter terminates the verification process, a ballot has already been registered in the ballot box, and this ballot corresponds to what the voter cast. This provides both the *Cast-as-Intended* and *Recorded-as-Cast* properties.

This works well for verifiability properties, but is not usable for properties like vote secrecy. These properties involve more complex proofs with *equivalence properties*. For vote secrecy, an equivalence property would state that given two scenarios, one where Alice votes 0, and one where Alice votes 1, an attacker should not be able to distinguish them. If a vote happens,

the attacker is unable to know if the vote is from the first or the second scenario. This kind of property can also be expressed in tools like Proverif to get formal proofs.

3.3 Limitations

Formal proof of the protocol is in itself absolutely useless if the software executing it has vulnerabilities. This includes checks done by the voting server to make sure that ballots are well-formed, or issues in the decrypting software that might make it possible to decrypt specific ballots without the consent of every authority.

It is very difficult to make sure that a software implementation correctly matches the formally verified protocol. And even if the logic of the implementation is correct, classical bugs like buffer overflow or use-after-free can still be decremental to the software security, allowing malicious actors to exploit the protocol.

Another important point with such formal proof are the security assumptions of the cryptography used. In the symbolic model these assumptions are written in the equational theory : The only way to derives the content of an encrypted message is to know the encryption key. But crypto-systems never have such clear-cut security properties, they can be subject for example to padding attack ([3]), or weaken the security of ciphertext if the same message is encrypted multiple times with the same key. These subtleties are not taken into consideration, and one must be careful when choosing the underlying cryptographic primitives for a protocol, even when proved in Proverif.

4 Fuzzing

Many tools and methods exists to test actual software implementation of a protocol. It ranges from simple unit-testing, to fully integrated End-to-End test, including audits and fuzzing, the subject of this section. While writing tests mostly detects expected bugs, the *fuzzing* approach tries to find bugs by brute-forcing unexpected comportments from the program under test (PUT).

Simply put, fuzzing is the process of repeatedly running the PUT with generated inputs in order to find bugs [18]. Fuzzers originally were made to find *crash case*, i.e., inputs that would make the PUT crash (as in [20]), but over the years many other bugs conditions were implemented in fuzzers.

4.1 Principle

The general working of a fuzzer is shown in the Algorithm 1. It consists of two parts, a first preprocess and the main *while* loop that does many *fuzz* runs.

The algorithm takes as input a set of parameters, and a timeout (t_{limit}), and outputs a set of possible bugs. A fuzzer also needs an internal *bug oracle* ($\mathcal{O}_{bug}$), that determines if a given execution contains a bug or not. This oracle could be the address sanitizer for example, which would mark execution as buggy if there are memory leaks, out of range access or use-after-free.

Preprocess The preprocess is the first step of the fuzzer. It takes the given parameters and makes a new set for the rest of the execution. For example, it might change the PUT to add information to the operations, or change the input seed.

Continue The `Continue` function can make the fuzzer terminates before the timeout happens. While most fuzzers continues the execution until the timeout to find as many bugs possible,

```
   input  : ℂ, t_limit
   output: 𝔹, a set of possible bugs
 1 𝔹 ← ∅;
 2 ℂ ← Preprocess(ℂ);
 3 while t_elapsed < t_limit ∧ Continue(ℂ) do
 4 |   conf ← Schedule(ℂ, t_elapsed, t_limit);
 5 |   test_cases ← InputGen(conf);
 6 |   𝔹′, execinfos ← InputEval(conf, test_cases, 𝒪_bug);
 7 |   ℂ ← ConfUpdate(ℂ, conf, execinfos);
 8 |   𝔹 ← 𝔹 ∪ 𝔹;
 9 end
10 return(𝔹);
```

Algorithm 1: General Fuzzing algorithm (from [18])

some white-box fuzzers (see Section 4.2) can terminate early if they are certain every possible value has been tested.

Schedule The `Schedule` function take the current set of parameters and extract one configuration for the fuzz iteration.

InputGen The `InputGen` function takes the fuzz configuration and returns the test cases that needs to be run.

InputEval The `InputEval` function runs the test cases with the set of parameters against the bug oracle. This has two important outputs :

- The set of bugs found with the execution of the test cases according to the bug oracle.
- Information about the executions. This can include code coverage, lines where the bugs are found, execution traces.

ConfUpdate Finally, the configuration is updated according to the new information from the executions.

This general algorithm can describe every fuzzers, but of course, there are many variations in the working and goals of fuzzerss.

4.2 The diversity in fuzzers

While fuzzers share the same framework, they can vastly differ in the specifics and in their goals. The firsts fuzzers made were *offensive* meaning they were made to find vulnerabilities (manly memory-based vulnerabilities) on already available binaries. Thus, this kind of fuzzer were *Black-Box fuzzers*, meaning that they did not have access to source code, just the binary that needed to be tested.

In reaction to the development of offensive fuzzers, software engineers started to fuzz their own software to find vulnerabilities before malicious actors. This made it possible to have *White-Box fuzzers*, in which, in addition to the comportment of the PUT, the fuzzer can get a lot more information with access to the source code, like code-coverage. White-Box fuzzing also means that it is possible to instrument the program at compile-time (with Asan for example)

instead of relying on dynamic instrumentation (with Valgrind for example), possibly improving performances and the quality of results.

In between white-box and black-box fuzzers is the category of *Grey-Box* fuzzer, containing any fuzzers that is not clearly either black or white-box. This includes fuzzers that don't have source-code but debug information in the binary, or fuzzers that exploit the underlying protocol executed by a software without knowing the actual source-code.

Finally, fuzzers differs in their *scope*, i.e., what they are fuzzing and what they are trying to find. A fuzzer made for Kernel module will be different from a fuzzer made for network-related software. Some fuzzers are exhaustive and prove results when they terminate, while other (most of them) just run for as long as possible and report all the bugs found, without providing any proof of security. This has a great incidence in the *Bug Oracle*. For a Kernel module, any memory issue is considered a bug, but for a network software things like authorization bypass must be considered.

5 Dolev-Yao-Model-Guided Fuzzing

Classical fuzzing tools can find issues in protocol implementation such as use-after-free bugs, but due to the nature of the mutation and of their bug oracle, they would not find *logical attacks*. To solve this, *puffin*[1] combines classical fuzzing methods with the DY model presented in Section 3.1 in order to both generate complex DY-trace that might trigger protocol violation, and a bug oracle able to detect violation of properties (as in Section 3.2) at the logical DY level.

This means that puffin will try to execute many protocols run on the PUT, and checks that the security claims still holds, and then apply mutation directly to the protocol trace, generating new inputs such as "change encryption key from x to y in this message", or "re-send the last message".

5.1 Principle

Puffin follows the general working of any fuzzer as described in Algorithm 1, but has some additional components in order to bridge the gap between the DY representations and the actual software programs that need to be tested. This is done with the *Mapper*, that maps abstract DY function to actual bitstring operations, and the *Harness*, that sends the correct inputs to the correct agents, and then inspect the internal state of the PUT to challenge the security claims.

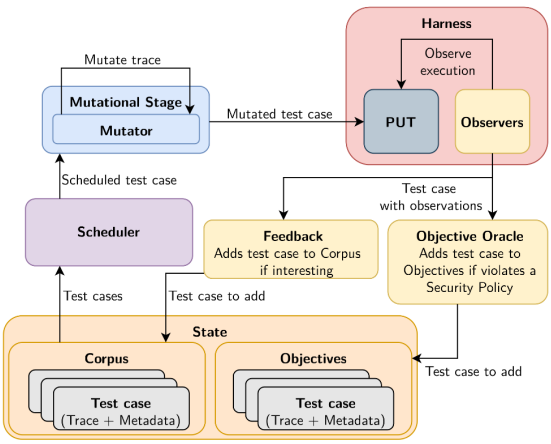

Figure 3: General architecture for Dolev-Yao-Model-Guided Fuzzing

The general working scheme of puffin is shown in Figure 3, where we find the same main stages of any fuzzer. However, a very important point is that, except in the "Harness" part (in red), puffin works with DY-trace (see Section 3.1). It is only at the last moment that the function symbols are interpreted against the cryptographic implementation of the fuzzed protocol.

This interpretation is done by the *Mapper*. The mapper takes as input a DY-term and output the corresponding bitstring that can be sent to the PUT. This part is protocol-dependent, but PUT-independent. Two implementation of the same protocol must use the same cryptography.

The part that calls the mapper is the *Harness*, and it is explained in more details in the next section.

The last specific part of the *puffin* fuzzer is about how cases are mutated. As *puffin* gets to work directly with DY traces, the mutations are directly done on them. The Table 1 shows the different mutations that can be applied on a DY trace.

Mutation	Description
Skip	Removes an action from a trace
Repeat	Repeats an action from the trace to the trace
Swap	Swaps two (sub-)terms in the trace
Generate	Replace a term by a random one
Replace-Match	Swaps two operators in the trace
Replace-Reuse	Replace a (sub-)term by another (sub-)term in the trace
Remove-and-Lift	Replaces a (sub-)term by one of its (sub-)term.

Table 1: Mutation for DY-fuzzing (from [1])

5.2 Harness

The Harness is the PUT-specific part of the puffin fuzzer. The Harness has two jobs in a fuzzing campaign : Sending the correct inputs to the correct agents, and getting results from the execution of said agents.

As an input, the Harness takes a DY-trace. The first thing done is to spawn one agent of the PUT per channel in the DY-trace. This allows to simulate things like an Adversary-in-the-middle attack between a client agent and a server agent. The Harness then sends, in the correct order, the respective terms to the agents corresponding to the channel. The terms pass through the mapper in order to give bitstrings to the PUTs.

The second job of the Harness is to interpret what is happening and adding it to the states. This is the *Observer* part of the Harness, that performs *Claim extraction* (claims as in Section 3.2. This part is very PUT-dependent, as it basically needs to know if some functions are called, and with which arguments. On most PUT, this data is exposed via logs or callbacks.

Once the claims are extracted, the Harness gives it to the *Objective Oracle* that determines if the given trace has violated a security property.

5.3 Results

Dolev-Yao-Model-Guided fuzzing has already been used to find vulnerabilities in protocol implementation. This has been done for TLS with *tlspuffin* in [1], and found new CVE on *wolfSSL* despite being software already under massive fuzzing campaigns.

The results on TLS proved that DY-fuzzing can find new protocol and memory related bugs in a protocol implementation. The approach using DY-traces allows to reach states that would otherwise be impossible with classical fuzzing, and can detect protocol based bugs.

However, at this time, DY-fuzzing has only been applied to simple client-server protocols. The work needed to be done to use this approach for more complex protocols (i.e., e-voting protocols) is described in the Section 6.

6 Using DY-Fuzzing on e-voting protocol

While DY-fuzzing has been proved to be effective for simple client-server protocol, it is not yet possible to just use this framework for the fuzzing of e-voting protocols.

First, e-voting protocols often involves multiples parties intercommunicating with each-other, making the fuzzing of the protocol as a whole much more difficult. Some security properties might be broken only when a big amount of legitimate voters do some specific actions, and current DY-fuzzing cannot find such attacks.

This comes from a bigger issue of representing the trust models in DY-fuzzing. A voting protocol is not only secure against an external attacker that intercept and modify messages. It must be secure against the possibility of some participants of the system being malicious. The administrator running the voting server might want to change the content of the ballot box, and he has a lot more power to do such than an external attacker. E-voting protocols clearly states their *threat-model* when discussing security, what component can be malicious and what are the consequences for the security properties.

And even in such scenarios, the *accountability* and *recoverability* of the voting system are very important factors. If a malicious internal actor of the voting system tries to break it, it could want to do so in an unaccountable way, meaning that it will not be possible to know that the fault came from this component. This also includes *Honest-but-Curious* component, that executes the election properly, while trying to learn as much as possible about what is happening.

In addition to that, as of now, DY-fuzzing can only detect trace-based property violation. This means that properties about vote secrecy cannot be captured by a fuzzing campaign. This is critical blind spot for e-voting, especially as implementation can leak and log several more metadata than what is necessary for the protocol, creating possibility for privacy leaks.

Bibliographie

[1] Max Ammann, Lucca Hirschi, and Steve Kremer. *DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing*. Cryptology ePrint Archive, Paper 2023/057. 2023. URL: https://eprint.iacr.org/2023/057.

[2] David Baelde et al. "The Squirrel Prover and its Logic". In: *ACM SIGLOG News* 11.2 (Apr. 2024). DOI: 10.1145/3665453.3665461. URL: https://inria.hal.science/hal-04579038.

[3] Romain Bardou et al. *Efficient Padding Oracle Attacks on Cryptographic Hardware*. Research Report RR-7944. INRIA, Apr. 2012, p. 19. URL: https://inria.hal.science/hal-00691958.

[4] Josh Benaloh. "Simple Verifiable Elections". In: *2006 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 06)*. Vancouver, B.C.: USENIX Association, Aug. 2006. URL: https://www.usenix.org/conference/evt-06/simple-verifiable-elections.

[5] Bruno Blanchet. "CryptoVerif: A Computationally Sound Mechanized Prover for Cryptographic Protocols". In: *Dagstuhl seminar "Formal Protocol Verification Applied"*. Oct. 2007.

[6] Bruno Blanchet, Martín Abadi, and Cédric Fournet. "Automated Verification of Selected Equivalences for Security Protocols". In: *Journal of Logic and Algebraic Programming* 75.1 (Feb. 2008), pp. 3–51.

[7] Federal Chancellery. *Federal Chancellery Ordinance on Electronic Voting*. 2022. URL: https://www.fedlex.admin.ch/eli/cc/2022/336/en.

[8] CNIL. *Projet de Recommandation : Sécurité des systèmes de vote par correspondance électronique (SVE)*. 2025. URL: https://www.cnil.fr/sites/default/files/2025-01/projet_de_recommandation_securite_des_systemes_de_vote_par_correspondace_electronique.pdf.

[9] Véronique Cortier, Alexandre Debant, and Pierrick Gaudry. *A privacy attack on the SwissPost e-voting system*. Talk given at RWC 2022. Presentation at https://iacr.org/submit/files/slides/2022/rwc/rwc2022/16/slides.pdf. 2022.

[10] Véronique Cortier, Alexandre Debant, and Pierrick Gaudry. "Breaking verifiability and vote privacy in CHVote". In: *Esorics 2025 - 30th European Symposium on Research in Computer Security*. Toulouse, France, 2025.

[11] Véronique Cortier and Pierrick Gaudry. *Le Vote Électronique*. Odile Jacob, 2022. ISBN: 9782415002206.

[12] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. "Belenios: A Simple Private and Verifiable Electronic Voting System". In: *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*. Ed. by Joshua D. Guttman et al. Cham: Springer International Publishing, 2019, pp. 214–238. ISBN: 978-3-030-19052-1. DOI: 10.1007/978-3-030-19052-1_14. URL: https://doi.org/10.1007/978-3-030-19052-1_14.

[13] Véronique Cortier et al. *A Practical and Fully Distributed E-Voting Protocol for the Swiss Context*. Cryptology ePrint Archive, Paper 2025/1625. 2025. URL: https://eprint.iacr.org/2025/1625.

[14] Véronique Cortier et al. “Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3783–3800. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/cortier.

[15] Alexandre Debant and Lucca Hirschi. *Reversing, Breaking, and Fixing the French Legislative Election E-Voting Protocol*. Cryptology ePrint Archive, Paper 2022/1653. 2022. URL: https://eprint.iacr.org/2022/1653.

[16] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[17] Sven Heiberg et al. “Improving the Verifiability of the Estonian Internet Voting Scheme”. In: *Electronic Voting*. Ed. by Robert Krimmer et al. Cham: Springer International Publishing, 2017, pp. 92–107. ISBN: 978-3-319-52240-1.

[18] Valentin J.M. Manès et al. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[19] Simon Meier et al. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701. ISBN: 978-3-642-39799-8.

[20] Barton P. Miller, Lars Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[21] Florian Moser et al. *A Study of Mechanisms for End-to-End Verifiable Online Voting*. 2024. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Cryptography/End-to-End-Verifiable_Online-Voting.pdf.

[22] Johannes Müller and Tomasz Truderung. “CAISED: A Protocol for Cast-as-Intended Verifiability with a Second Device”. In: *Electronic Voting*. Ed. by Melanie Volkamer et al. Cham: Springer Nature Switzerland, 2023, pp. 123–139. ISBN: 978-3-031-43756-4.

[23] SwissPost. *SwissPost E-Voting Bug Bounty Program*. 2025. URL: https://yeswehack.com/programs/swiss-post-evoting.