SYMBEXCEL: Automated Analysis and Understanding of Malicious Excel 4.0 Macros

Nicola Ruaro[†], Fabio Pagani[†], Stefano Ortolani[‡], Christopher Kruegel[†], Giovanni Vigna[†]
[†] University of California, Santa Barbara, [‡] VMware
{ruaronicola, pagani, chris, vigna}@ucsb.edu, ortolanis@vmware.com

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833765

***Abstract*—Malicious software (malware) poses a significant threat to the security of our networks and users. In the ever-evolving malware landscape, Excel 4.0 Office macros (XL4) have recently become an important attack vector. These macros are often hidden within apparently legitimate documents and under several layers of obfuscation. As such, they are difficult to analyze using static analysis techniques. Moreover, the analysis in a dynamic analysis environment (a sandbox) is challenging because the macros execute correctly only *under specific environmental conditions* that are not always easy to create.**

This paper presents SYMBEXCEL, a novel solution that leverages symbolic execution to deobfuscate and analyze Excel 4.0 macros automatically. Our approach proceeds in three stages: (1) The malicious document is parsed and loaded in memory; (2) Our symbolic execution engine executes the XL4 formulas; and (3) Our Engine concretizes any symbolic values encountered during the symbolic exploration, therefore evaluating the execution of each macro under a broad range of (meaningful) environment configurations.

SYMBEXCEL significantly outperforms existing deobfuscation tools, allowing us to reliably extract Indicators of Compromise (IoCs) and other critical forensics information. Our experiments demonstrate the effectiveness of our approach, especially in deobfuscating novel malicious documents that make heavy use of environment variables and are often not identified by commercial anti-virus software.

I. INTRODUCTION

Despite advances in computer security, cybersecurity threats are still on the rise. Ransomware attacks, for instance, represent one of the most devastating threats of the past few years, expected to inflict more than $20 billion in damages in 2021 alone [51]. Therefore, the detection and forensics analysis of security threats is critical for protecting users and organizations from malicious actors.

Attackers leverage different techniques to infect a victim's system, ranging from exploiting vulnerabilities in Internet-facing services to phishing campaigns stealing the victim's credentials. A popular approach is to abuse Microsoft Office documents as carriers for malicious macro code. The attacker usually sends these malicious documents as email attachments, and deceives the victim into opening them and executing the malicious payload.

One of the most well-known examples of macros utilized for malicious purposes are Visual Basic for Applications (VBA) macros, which security analysts have studied extensively in the past [34], [48], [49], [50]. However, since security researchers have contributed numerous open-source tools to analyze and deobfuscate VBA macros, malware authors have recently shifted their attention to a lesser-known macro format: *Excel 4.0* macros [28], [67]. Excel 4.0 macros, or XL4 macros[1], are a 30-year-old feature of Microsoft Excel that allows one to encode a series of operations into an Excel file. While this feature originates as a precursor of VBA macros, it is similarly powerful and still used in corporate settings.

In the past few years, malware campaigns using XL4 malware have been deployed at scale, and infections related to this threat have increased [28], [66], [67]. However, for a series of reasons, the XL4 malware ecosystem remains largely unexplored, due to a number of challenges associated with the analysis of XL4 macros.

First, Microsoft Excel supports several macro formats with complex specifications [56]. Malware authors always seek new ways of creating spreadsheets that break static parsers (used by analysis tools) while remaining compliant with the actual parser shipped with Excel. Second, malware authors have deployed a series of evasion techniques that hinder both static and dynamic analysis. In particular, heavy obfuscation techniques have been recently introduced, effectively preventing every available tool from correctly analyzing these samples. These techniques use information about the execution environment both to detect sandboxes and to deobfuscate the malicious payloads. Third, malware analysts can only rely on limited tooling when inspecting a potentially malicious document. When these tools cannot process a sample, the analyst has to resort back to manual analysis, which consists in opening the file in Excel and manually stepping through the XL4 formulas. However, such a manual approach is complicated by the layers of obfuscation techniques included in XL4 malware, and by the large number of formulas executed (often in the range of thousands of formulas).

Correctly deobfuscating XL4 malware remains a critical task. On the one hand, in post-mortem scenarios, it is essential to understand what malicious actions have been performed. On the other hand, to prevent future infections, it is essential to extract any indicator of compromise (IoC).

In this paper, we study the ecosystem surrounding Excel 4.0 malware, and we propose SYMBEXCEL, a novel system to automatically analyze advanced XL4 malware samples. The core component of our system is a symbolic execution engine for XL4 macros. Using symbolic execution, we can automatically infer the "correct" values of any environment variable—i.e., the values that lead to the deobfuscation of the malicious payload. Overall, our approach also provides valuable insights into malware's interactions with and dependencies on the environment. Moreover, our symbolic engine follows multiple paths of execution (i.e., states) during the symbolic exploration, and is, therefore, able to capture every possible behavior of the malicious samples. For instance, malware samples often try to connect and download a second-stage payload from multiple servers, in case one of them becomes unreachable. In these cases, thanks to symbolic execution, we are able to explore all the possible execution paths, and therefore extract the address of every infected server.

Another important aspect of our system is that it can handle symbolic Excel 4.0 formulas. Excel malware often uses information retrieved from the environment, which is represented as symbolic values in our system, to decrypt and execute Excel formulas. Given the combination of two novel techniques presented in this paper, namely *observers* and *smart concretization*, SYMBEXCEL is able to concretize the decrypted formulas and to efficiently continue

[1] We will use the terms Excel 4.0 macro and XL4 macro interchangeably.

DOI 10.1109/SP46214.2022.00105

the symbolic exploration. Finally, to avoid reimplementing the entire Excel engine in our system, we propose a technique called *delegations* to offload the execution of certain formulas to Excel.

We evaluate SYMBEXCEL on two different datasets of XL4 samples, one that contains 5,697 publicly available samples, while the other contains 18,840 private samples. Our datasets cover a period spanning more than eight years—albeit malicious actors started to leverage Excel 4.0 macros on a large scale only in the past few years. SYMBEXCEL performs significantly better than existing tools, deobfuscating correctly 23,931 (instead of 12,375) out of 24,537 samples. Moreover, if we restrict our analysis to the samples that make heavy use of obfuscation and machine-specific variables, SYMBEXCEL can deobfuscate correctly 7,239 (instead of 410) malware samples.

In summary, our paper makes the following contributions:

- We present SYMBEXCEL, a symbolic execution engine for Excel 4.0 macros that can deobfuscate advanced malicious documents.
- We evaluate our system on a dataset of more than 25,000 malicious samples, covering a period of more than 8 years.
- We present an in-depth overview of the most common evasion technique adopted by malware authors, and a study of the malicious behaviors observed.

We believe that our results help to shed light on the XL4 malware ecosystem, and that SYMBEXCEL will help incident response and malware analysts to understand XL4 malware successfully. To foster more research in this field, we make our code available at https://github.com/ucsb-seclab/symbexcel.

II. BACKGROUND

Excel File Formats. Microsoft Excel supports tens of different file formats, but only four of them can contain Excel 4.0 macros and are routinely used to deliver Excel 4.0 malware. For this reason, in this paper we will only focus on the following file formats: *Excel 97 - Excel 2003 Workbook* (`.xls`), *Excel Binary Workbook* (`.xlsb`), *Excel Workbook* (`.xlsx`) and *Excel Macro-Enabled Workbook* (`.xlsm`). The first two formats are binary file formats, also known as Binary Interchange File Format 8 (BIFF8) and Binary Interchange File Format 12 (BIFF12), respectively. Microsoft released these format specifications in 2008 [44], [45]. On the other hand, `xlsx` and `xlsm` files are based on XML. Despite the differences between these formats, from an application standpoint, every Excel file consists of a *workbook* and one or more *spreadsheets*. A workbook is a collection of spreadsheets, which in turn contain the cells where formulas and values are stored. A spreadsheet can be further classified as a *macro sheet* or as a *worksheet*, with the only difference being that *macro sheets* are the only type of spreadsheet that can contain Excel 4.0 macros. Finally, a workbook can contain one or more globally defined variables called *defined names*, which have an associated value and are shared across the spreadsheet.

Excel 4.0 Macros. This feature of Microsoft Excel was released in 1992, and it was soon widely adopted in numerous organizations. Interestingly, despite the introduction of VBA macros as a replacement for XL4 macros, the latter are still supported by the latest version of Microsoft Excel. Excel 4.0 macros are a super-set of the traditional Excel functions [43] and offer a large set of functions to interact with an Excel workbook and with the external environment.

To avoid any confusion, in the rest of this paper, we will refer to the distinct Excel 4.0 macro functions (e.g., `EXEC`, `CHAR`) as *functions*. We will instead use the term *formulas* to denote expressions that always start with an equal sign and use one or more functions (e.g., `=EXEC("calc.exe")`). Finally, we will refer to sequences of formulas that are stored in a *macro sheet* as *macros*.

In particular, unlike traditional Excel functions (such as `SUM`) used in spreadsheets, Excel 4.0 macro functions have access to the Windows API and can be used to interact with the operating system. For example, the formula `=FILES(directory)` lists all the files in a given directory, and an attacker can use `=EXEC(program)` to launch an external program. In practice, Excel 4.0 macros are nothing less than a sequence of Excel 4.0 macro formulas, just like binary programs are a sequence of instructions.

XL4 macros cannot reside in regular worksheets. Instead, they must be stored in specific Excel 4.0 macro sheets, one formula per cell. The execution of XL4 macros starts from a cell—called *AutoOpen*—and continues executing the underlying cells until either a terminating function is encountered (i.e., `=HALT()`) or until a control-flow transferring function is executed (e.g., `GOTO(CELL)`). In the latter case, the execution continues with the code stored in the target cell. Using the functions `FORMULA` and `FORMULA.FILL`, Excel 4.0 malware can also generate formulas on-the-fly, and store them in a macro sheet for later execution.

Execution Environment. As mentioned previously, Excel 4.0 macros can interact with the operating system. Advanced XL4 malware makes heavy use of this capability and frequently checks the environment to fingerprint sandboxes and to deobfuscate additional layers of the malicious payload. In this paper, we define as *execution environment* every information that is not directly contained in a workbook. For example, XL4 malware often uses the `GET.WORKSPACE` function to access information from the execution environment, such as the total memory available to Microsoft Excel, the name of the underlying operating system, and the version of Microsoft Excel. Similarly, any information about a window—such as its name, size, and position—can be accessed using the `=GET.WINDOW` function. Finally, some samples execute the function `NOW` to retrieve the current timestamp and use this information to deobfuscate a malicious payload.

Current Defenses. Excel 4.0 malware is a threat that has surged only recently. Hence, only a few defenses are currently available. To the best of our knowledge, the only runtime defense designed explicitly for Excel 4.0 malware is the Microsoft AMSI integration, released by Microsoft in 2021 [47]. The Microsoft AMSI integration deploys hooks inside a running Excel process and, whenever certain Excel 4.0 functions are executed, it forwards the macro to an external antivirus engine, which decides whether it is malicious or not.

Another approach, mainly used for deobfuscation and post-mortem analysis of malicious documents, is based on the emulation of Excel 4.0 functions. This analysis technique works by interpreting all the instructions executed by the malware sample without running the sample itself on a physical machine. An example of such an analysis tool for XL4 macros is *XLMMacroDeobfuscator* [53], an

① **Sandbox fingerprinting**

```
[A1] =IF(GET.WORKSPACE(19), SET.VALUE(K1, K1+1),
                            SET.VALUE(K1, 0))
[A2] =IF(GET.WORKSPACE(42), SET.VALUE(K1, K1+1),
                            SET.VALUE(K1, 0))
[A3] =IF(K1<2, CLOSE(TRUE), )
```

② **Key calculation and payload decryption**

```
[A4] =FORMULA(DAY(NOW())+K1, K2)
[A5] =FORMULA(CHAR(B1-K2) & .. & CHAR(B20-K2), C1)
```

③ **Payload execution**

```
[A6] =GOTO(C1)

[B1-B20] DL_LJ/)wv~lyzolss55)0

[C1] =EXEC("powershell..")
[C2] =HALT()
```

Fig. 1: Example of a malicious Excel 4.0 macro payload.

open-source project that we leverage as the foundation for our work. In general, these emulator-based tools parse the malicious Excel file, find the correct entry point, and then emulate all the instructions, trying to replicate the original Excel 4.0 macro behavior.

A significant limitation of these tools is using a default configuration to model the environment. Some samples might not show their true behavior when executed under the default environment configuration, and inferring *a priori* the environment expected by the malware is challenging. For example, some samples run their malicious payloads only when executed on a specific date or when the width of the Excel window is larger than a particular threshold.

Recently, Leibovich and Ciuraru presented an approach based on machine learning to identify Excel 4.0 malware families [41]. The authors use *t-SNE* and *k-means* with a set of 253 features, spanning from the length of the strings contained in a document to the count of occurrences of each Excel 4.0 function. While this approach can be used to detect variations of previously seen malware samples, it does not provide any actionable forensics information on samples that were not previously observed.

Symbolic Execution. Symbolic execution is a program analysis technique that executes the program in the abstract domain of symbolic variables, forking the execution after every conditional instruction and keeping track of all the constraints introduced during the execution. For example, when reading an integer environment variable `X`, this variable is initially unconstrained and can assume any possible integer value. However, after the execution of a conditional instruction with a guarding condition `X`$\geq$`0`, the analysis will fork into two new branches and constrain the variable's value to be either positive (where the constraint is `X`$\geq$`0`) or negative (where the constraint is `X<0`). This type of analysis allows one to determine the inputs that trigger the execution of a particular branch in the program.

III. Motivation

Although Excel 4.0 macros offer several functions that malware authors can trivially use to infect a system, malicious payloads have evolved notably over time. In particular, malware authors have deployed a series of evasion techniques that utilize the execution environment to hinder static and dynamic analysis.

Figure 1 shows an example of these techniques. The first two formulas (cells A1 and A2) call the function `GET.WORKSPACE`, which retrieves some information about the execution environment. In particular, the effect of these two formulas is to check whether the machine has audio and mouse capabilities and—if these capabilities are detected—to increment the value of the cell K1. The formula in cell A3 checks if the value of K1 is less than 2, and it aborts the execution when this condition is true. The macros in cells A4 and A5 are responsible for decrypting the malware payload. In particular, the formula in cell A4 adds the current day of the week to the value of cell K1, and stores this result in cell K2. The malware then subtracts the value of K2 to the characters from the range B1–B20, which are then concatenated and stored in C1. After this operation, cell C1 will contain the decrypted malware payload, which, in our example, calls the `EXEC` function to execute a command using *powershell*. Finally, the formula in A6 transfers the control flow to the decrypted payload (stored in C1) before halting the execution (cell C2).

Concrete Analysis. This analysis technique works by interpreting each formula of a malicious document, using a default execution environment. The example in Figure 1 clearly shows the limitations of this approach. In particular, before the analysis, we do not know whether the malware expects to run in an environment where mouse and audio capabilities are present. These anti-analysis checks are generally used to detect whether a sample is running in a sandbox environment or not. For our example, we could create an environment that pretends that basic capabilities are present. However, we would not be able to easily determine the correct date that the malware expects. This date is retrieved using the formula `NOW()` (cell A4 in the previous example) and is used to decrypt the malware payload (cell A5). Using a "wrong" value will generate an invalid payload, which will hide the real behavior of the malware. A similar technique has been observed in real malware samples, and it is intended to hinder any dynamic analysis that runs on a day different than the intended one [28].

In general, to overcome the problems related to the unknown environment, concrete analysis can be coupled with forced execution. This technique forces the execution to take different branches on conditional instructions, using brute-force to iterate over different environment variables. This technique allows one to partially side-step the environment configuration problem, but not without limitations. First, forced execution only allows to bypass simple checks (conditions), but it does not guarantee the correct environment configuration when forcing the execution down a particular branch. In our running example of Figure 1, if the value of K1 is equal to 1 when executing the formula in A3, forced execution can divert the execution towards the *false* branch, and avoid the `CLOSE` function. Even though this diversion will make the execution reach the deobfuscation routine in cell A5, the value of `K1` will be wrong (leading to an incorrectly decrypted payload in C1). Similarly, brute-forcing requires identifying the subset of relevant environment variables and finding an efficient strategy to triage several combinations of their values. As we show in the evaluation in Section V, this approach can be useful to test different values

of the date used in cell A4. However, when the malware sample uses a more complex environment configuration, the search space quickly increases in size, and this approach becomes infeasible.

Symbolic Analysis. As described in the paragraphs above, concrete analysis has several limitations, mainly related to its inability to reason about unknown environment variables. On the other hand, symbolic execution is a suitable technique to keep track of how environment variables are generated, propagated, and used during execution. This information can be used to reason more formally about the environment, and to represent any possible value of the environment variables. For example, when SYMBEXCEL executes the `GET.WORKSPACE` function in cell A1, symbolic execution postpones the decision on the concrete value that this function should return, and instead returns a *symbolic variable*. Then, since this symbolic variable is used as a condition in an `IF` function, symbolic execution forks the execution and generates two separate states: one that follows the *true* branch, while the other follows the *false* branch. The memory of the states will be updated accordingly—i.e., the first state contains `K1=1` while the second one contains `K1=0`. A similar process is then repeated for the formula in cell A2. Only one of the four generated states will contain `K1=2`, and it will therefore reach the deobfuscation routine, while all the other three states will be terminated after executing the formula in A3. Moreover, since we consider the date as part of the environment, also cell K2 will contain a symbolic variable, and the decrypted formula stored in C1 will also be a symbolic expression. When the execution will reach C1, our system, using a *smart* concretization technique, will concretize the symbolic variable and execute the malicious payload.

IV. SYMBEXCEL

This section discusses the architecture of SYMBEXCEL, our symbolic execution engine to analyze Excel 4.0 macros. Figure 2 provides an overview of the different components of our system. In order to analyze a document containing Excel 4.0 macros, SYMBEXCEL first needs to parse the document itself. To achieve this goal, we use a *Loader* component (either the *Static Parser* or the *COM Loader*) that understands the underlying file format (e.g., `BIFF8` [56]) and extracts the information that is needed to run the sample in our analysis environment. The second component of SYMBEXCEL, which we call the *Execution Engine*, implements the symbolic analysis engine. This is the core component of our system, which is responsible for interpreting the formulas contained in the document—using symbolic variables to model the execution environment—and for guiding the symbolic exploration. The third and last component of SYMBEXCEL is the *Solver Backend*. This component is a wrapper around an SMT constraint solver, and is responsible for checking the satisfiability of the collected constraints (for example, when executing a conditional instruction) and for translating expressions from the symbolic to the concrete domain. The output of our system, once the symbolic exploration terminates, is a report containing any security-relevant formulas (SRFs) observed during the execution, which can be parsed to extract IoCs such as filenames, URLs, shell commands, registry keys, cryptographic hashes of dynamically generated files, etc. For example, in Figure 1, the formula `EXEC("powershell..")` can be trivially parsed to extract the shell command `"powershell.."`. In this paper, we define the following formulas as SRFs: `EXEC`, `CALL`, `REGISTER`, `FOPEN`, `FWRITE`, `FWRITELN`.

In the following paragraphs, we describe the three main components of SYMBEXCEL in more detail, explaining our design and implementation choices.

A. Loader

This component is responsible for parsing the Excel document and extracting all the information needed to start the analysis. Such information includes the name and content of all the sheets in the workbook, the analysis entry points, the defined names, the values and formulas in each cell, and the properties of each cell (e.g., font information, background color). We then import this information into our analysis environment by creating an *Execution Engine* instance and by initializing its memory and environment to reflect the contents of the original Excel file.

SYMBEXCEL can load an Excel document file using either a *Static Parser* or the *COM Loader*. The static parser leverages public knowledge [44], [45] about the structure of the `BIFF` and `XML` file formats to parse an Excel document. This approach makes the loader faster, but at the same time, less robust. As described in more detail in Section V, correctly implementing an Excel file parser is inherently hard, and malicious actors are routinely finding new ways to break such analysis tools' parsers while preserving the validity of the file with respect to Excel. An example of a statically implemented loader is the Python library called *xlrd2* [57], an open-source and regularly maintained project that we improve and use in this paper to parse malicious Excel files.

An alternative approach to parse and load such files is based on the Microsoft COM functionality[2], which can be used to directly interface with Excel. This approach allows deferring most of the work to the Excel parser implementation, which is inherently robust and faithfully mirrors the outcome of a normal execution scenario. Our COM Loader uses the Microsoft COM interface to load the Excel file into a running instance of Excel. It is, therefore, more resistant to some of the evasion techniques used in current Excel 4.0 malware samples (such as *xlrd2* parsing confusion, as discussed in Section V-B), albeit an order of magnitude slower than the static loader—on average, the loading time with the *Static Parser* is around 3 seconds, while the *COM Loader* can take as much as 30 seconds.

Entry point. Excel malware has different ways to start the execution of malicious Excel 4.0 macros. For this reason, a crucial piece of information extracted by the loader is the *entry point*, which is used by the Execution Engine to start the analysis. The first category of entry points are related to built-in functionalities of Excel 4.0 macro sheets. For example, the *Auto_Open* label specifies a macro that is automatically executed when the macro sheet is opened. Quite similarly, *Auto_Close*, *Auto_Activate* and *Auto_Deactivate* execute a macro when the spreadsheet is closed, when a workbook is activated and when a workbook is deactivated, respectively. Extracting these triggering macros is generally straightforward, since they are stored under well-known (constant) names.

[2]The Microsoft Component Object Model (COM) was introduced by Microsoft in 1993 and is a platform-independent, object-oriented binary-interface standard for creating binary software components that can interact with each other.

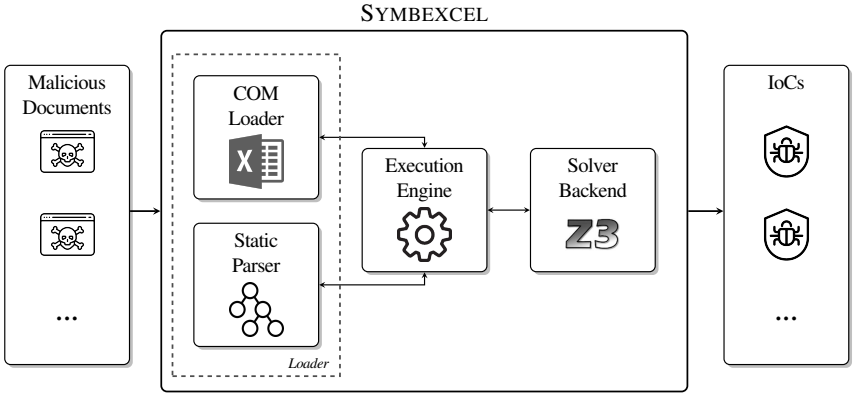

Fig. 2: System overview of SYMBEXCEL.

On the other hand, some of the latest malware samples first execute *VBA* code that, in turn, triggers the execution of Excel 4.0 macro. Handling such cases in a generic way would not only require a fully-functional VBA engine, but also the logic to handle the interactions between VBA code and Excel 4.0 macros. For this reason, SYMBEXCEL currently extracts Excel entry points only from single-statement VBA functions that contains the *Application.Run* method. Specifically, SYMBEXCEL extracts the VBA code using *oletools* [39], and then parses it to extract the Excel entry points. Since we did not observe any obfuscation in these VBA functions, we use a regular expression to extract the Excel 4.0 entry point.

B. Execution Engine

This component is responsible for orchestrating the symbolic exploration of a sample. In particular, the execution engine parses and dispatches the Excel 4.0 formulas to the proper function handlers, forks the execution when a conditional formula is executed, and invokes the solver backend to concretize symbolic values. The implementation of our execution engine builds on the open-source implementation of *XLMMacroDeobfuscator* [53]. The most important additions we made to this tool are related to symbolic execution: from supporting the creation and propagation of constraints to keeping track of multiple execution states. Moreover, we improved the formula parser to support Unicode characters, and we also implemented new function handlers.

Formula Parsing. In order to interpret an Excel 4.0 formula, SYMBEXCEL first parses the formula string from the target cell and generates an Abstract Syntax Tree (AST). The AST is a representation of the formula that makes its syntactic structure explicit. The formula parser used by our system is based on extended Backus–Naur form rules (EBNF), which describe the syntactic structure of the XL4 grammar. SYMBEXCEL implements several function handlers each implementing a non-terminal symbol of the XL4 grammar. Since each function handler replicates the behavior of a function on the execution state, after parsing the XL4 formula into an AST, we then walk the AST and dispatch the execution to one or more function handlers in a recursive descent fashion.

Execution States. When an Excel macro does not use any symbolic value, a *single* execution state can be used to represent the execution of the macro. The execution state is created at the beginning of the analysis, and the same state will be used until the macro terminates, since every formula will have only one valid successor.

On the other hand, when executing a conditional instruction (e.g., `IF`) that operates on at least one symbolic variable in its condition, a state can have *two or more* valid successors. For example, the `IF` function in cell A3 of Figure 1 uses a symbolic variable (K2) in its condition, and both branches are potentially valid since such symbolic variable is unconstrained. To handle these cases, the Execution Engine must duplicate the state and follow one or both branches, depending on the state's constraints. The results of this duplication process are new *execution states*, which follow different execution paths and eventually explore the macro's entire behavior.

Since these execution paths are independent of each other, every state contains its own copy of the memory, environment configuration, and constraints:

1) The **memory** holds the values and formulas contained in cells, the information regarding cells' properties (e.g., font information), and the *defined names*.
2) The **environment** contains accessory information that is not directly stored in a workbook. For example, the window height and the current operating system version are both environment variables. Malware authors use environment variables for sandbox detection and evasion. Therefore, depending on the environment, the same malware sample can show different behaviors. For this reason, we associate a symbolic variable to each of these environment variables, allowing us to explore every possible malware behavior.
3) The **constraints** characterize the malware behavior and are propagated from one state to its successors during the execution. For example, a possible constraint is `GET.WORKSPACE(14)>390`, which constrains the symbolic variable representing the window height to be greater than 390.

Function Handlers. The function handlers replicate the original behavior of a formula. The handlers can thus modify the execution state by updating the memory, accessing the environment, and adding new constraints:

1) **Updating the memory:** After executing an XL4 formula, SYMBEXCEL updates the memory of the state to store the formula's result and to reflect any side-effects in the state's context. For example, the formula in cell A1 of Figure 1, updates

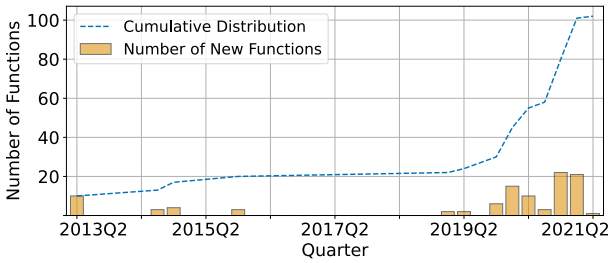

Fig. 3: Number of functions observed over time.

the value in cell K1 (when executing the `SET.VALUE` macro) and writes either `TRUE` or `FALSE` in cell A1, depending on which branch is executed.

2) **Accessing the environment:** As mentioned, XL4 malware samples use the environment values for sandbox detection and fingerprinting. For example, when executing the formula `=FORMULA(DAY(NOW())+K1, K2)` from our running example (Figure 1, cell A4), SYMBEXCEL starts by calculating the result of the expression `DAY(NOW())+K1`. Thus, the `NOW` function will access the environment to read from the execution state's context the symbolic value associated with the current time. Finally, the `FORMULA` function will write the expression's value to the destination cell K2. As a result, the symbolic variable is propagated from the environment to memory and will be used later in the execution for further computation.
3) **Generating new constraints:** As previously explained, when executing a conditional instruction with a symbolic operand, SYMBEXCEL will fork the execution state, update the states' contexts, and generate the correct guarding constraints. For example, the formula `=IF(K1<2, CLOSE(TRUE), )` (Figure 1, cell A3) will generate the two guarding constraints `K1<2` and `K1>=2`, which will be added to the respective states.

Delegations. In order to make our approach more resilient, we implement an additional mechanism that allows SYMBEXCEL to *delegate* the execution of an XL4 formula to Excel. The Excel 4.0 function reference [58] contains hundreds of functions, and implementing each of them inside SYMBEXCEL would basically require reproducing the entire Excel formula engine. Moreover, as seen in Figure 3, the number of functions used by the malware authors increases dramatically over time. For this reason, when our system tries to execute a function that is not implemented in our engine, we offload the execution to the COM server. In particular, SYMBEXCEL uses the COM functionality to communicate with a running instance of Microsoft Excel, and it synchronizes the current state context, which includes the content of the spreadsheets, macro sheets, and the defined names. It then executes the target formula, and finally, it fetches the resulting context back into our execution engine. This delegation mechanism allows our approach to scale even when some of the Excel functions used by a sample are not implemented in our execution engine.

Currently, SYMBEXCEL can only delegate instructions with concrete arguments. However, since only a subset of the formulas used in current malware samples has symbolic arguments, this technique dramatically reduces the implementation overhead when threat actors start using a new formula.

```
[A1] =FORMULA((GET.WORKSPACE(14)>390)+84, B1)
[A2] =FORMULA("=HAL" & CHAR(B1) & "()", C1)
[A3] =GOTO(C1)
```

Fig. 4: Execution of a symbolic expression (stored in cell C1).

Summary. The *Execution Engine* is the core component of our system: it parses each formula during the execution, generates the corresponding AST, and invokes the function handlers to reproduce the original formula's behavior. When one of the AST nodes is symbolic (i.e., refers to an environment variable), the function handlers will propagate this symbolic value and generate a new symbolic expression.

C. Solver Backend

The final component in our architecture is the *Solver Backend*, which is a custom wrapper around the Z3 SMT solver [26]. This component allows SYMBEXCEL to accumulate the constraints associated with an execution state, check their satisfiability, and concretize symbolic expressions.

SYMBEXCEL can concretize the path constraints to allow multi-path exploration, as well as the symbolic memory addresses and formulas to enable the analysis of self-modifying macros. Take as an example the formula `=FORMULA(..., C1)` from Figure 4, cell A2. The `FORMULA` function will evaluate the expression specified as the first argument and write its result into the destination cell, specified as the second argument. In this case, the corresponding handler will concatenate the string `"=HAL"` with the symbolic character stored in cell B1, append the string `"()"`, and finally write the result into cell C1. The formula `=GOTO(C1)` will then redirect the execution to cell C1, which contains a symbolic expression.

Since a symbolic expression can have multiple possible concrete values, SYMBEXCEL will concretize the symbolic expression and generate a set of concrete solutions that our symbolic execution engine can execute. Translating an expression from the symbolic domain to the concrete domain allows us to continue executing the malware sample, exploring all its possible behaviors.

As mentioned before, there are often multiple concrete solutions to a symbolic expression. The expression used in cell A1 of Figure 4 is a symbolic expression based on the environment variable returned by `GET.WORKSPACE(14)`. However, such variable is a symbolic integer variable with 2^{32} concrete solutions. Therefore, after executing cell `A3` and transferring the execution to the symbolic expression stored in cell `C1`, a naïve concretization strategy would fork 2^{32} execution states, overloading our symbolic execution engine. For this reason, we implement two optimizations that make our concretization strategy more efficient: *Observers* and *Smart Concretization*.

Observers. To make constraint solving more practical, we introduce additional symbolic variables during the symbolic exploration. An observer variable is an intermediate variable that represents a symbolic sub-expression. In particular, when we execute a symbolic comparison operation, a symbolic boolean operation, or when handling an `IS_NUMBER` formula on a symbolic string index (e.g., `ISNUMBER(SEARCH(...))`), we represent the resulting Boolean expression with a freshly created symbolic variable. This process is crucial when our system has to concretize a symbolic formula, since it dramatically reduces the concretization space.

For example, we assume that SYMBEXCEL needs to retrieve all possible concrete models associated with the symbolic expression in cell C1. When concretizing the symbolic expression `(GET.WORKSPACE(14)>390)+84`, our solver backend recognizes that this expression uses a symbolic integer variable. As a result, the number of possible models is 2^{32}—i.e., all the possible values associated with the symbolic integer variable `GET.WORKSPACE(14)`. On the other hand, if we strategically introduce a symbolic Boolean variable `OBSERVER_1` to represent the expression `GET.WORKSPACE(14)>390`, then the symbolic expression becomes `OBSERVER_1+84`. As a result, this expression can have only two concrete solutions, 84 and 85, and our solver backend can concretize the expression without additional overhead.

Smart Concretization. Even after introducing one or more observer variables, it is possible to have many concrete solutions associated with a symbolic expression. To limit the number of generated states, SYMBEXCEL uses the XL4 grammar to decide whether a concrete string is a valid formula or not. In other words, after retrieving every concrete model of a symbolic expression, we use this grammar as an oracle to filter any invalid formula. In the previous example in Figure 4, the two possible concretizations of the string stored in cell C1 are `"=HALT()"` and `"=HALU()"`. While the first concretization represents a valid XL4 formula, the second one is invalid and thus is discarded by SYMBEXCEL. This smart concretization strategy is without loss of generality, since Excel also aborts the execution when it encounters an invalid formula. In other words, malware authors cannot deceive our system into discarding their otherwise legitimate payload by intentionally using an invalid formula.

V. EVALUATION

We combine two different datasets of malicious Excel 4.0 macros to evaluate our approach. The first one, which we will refer to as the *public* dataset, is a collection of 5,697 samples retrieved from VirusTotal (VT). The second dataset, which we will refer to as the *private* dataset, contains instead 18,840 samples that are not publicly available and that were collected by a security vendor. Overall, these two datasets contain 24,537 malicious samples. Using the submission date, we estimate that our dataset covers more than eight years, since the earliest observed sample was submitted to VT in March 2013 and the latest observed sample was submitted in July 2021.

In the following sections, we describe the ecosystem behind Excel 4.0 malware, how it evolved over time, and how malicious actors continuously find new ways to hinder the analysis of XL4 macros. In particular:

- We present the results of our comparison with other publicly available tools. Specifically, we compare SYMBEXCEL with *XLMMacroDeobfuscator* [53], the state-of-the-art open-source concrete analysis tool for Excel 4.0 macros.
- We study the mechanisms used to trigger the execution of Excel 4.0 macros, and the most common evasion techniques that malware samples use to prevent the parsing of Excel files.
- We study the security-relevant formulas used by malware samples to execute malicious payloads, and the IoCs extracted from these formulas.
- We study the evolution of Excel 4.0 malware samples over eight years and present the results of our temporal study.

A. Approach Comparison

To assess the effectiveness of our approach, we compare our system with the state-of-the-art concrete deobfuscation tool *XLMMacroDeobfuscator* (version 2.0, released in November 2021).

The results of this comparison are presented in Table I where, for each tool, we report the number of samples that are successfully analyzed. We consider a sample to be successfully analyzed when the macro executes at least one of the following three functions: `EXEC`, `CALL`, or `REGISTER`. Malicious XL4 macros typically use these functions to implement the malware behaviors, such as calling a *WinAPI* function to download a file from a remote server, or to execute an external program. When one of these functions is encountered, we assume that the deobfuscation stages have been successfully executed and that the environment configuration is adequate to reveal the malicious behavior.

In Table I, we have divided the samples into two categories, *Concrete* and *Symbolic*, depending on whether or not a sample reads any information from the environment that is represented with a symbolic variable (e.g., the size of the window or the current day). This allows us to show the difference in the analysis results based on whether the execution of a sample relies on external inputs.

We also run SYMBEXCEL with *two* different configurations: one where the malicious documents are parsed with an improved version of the *xlrd2* static parser (SYMBEXCELSTATIC), and one where this task is offloaded to our COM Server (SYMBEXCELCOM). The results for SYMBEXCEL reported in the top part of the table show the union of the results for these two configurations. For our experiment, we also group similar samples together by using the sequence of executed formulas as an equivalence metric and presenting the results in terms of structurally distinct samples (right side of Table I).

The first key result presented in Table I is that SYMBEXCEL—in either configuration— deobfuscates significantly more samples than *XLMMacroDeobfuscator* (23,931 compared to 12,375 samples). Our better results are due to a number of factors. The first difference between the two approaches is that SYMBEXCEL's execution engine currently implements 103 function handlers, while *XLMMacroDeobfuscator* supports 73 functions. Moreover, using the COM server to load and parse the samples allows SYMBEXCEL to sidestep some of the evasion techniques that are used to interfere with the static parsers, as we will discuss in-depth in the *Evasion Techniques* section. Finally, SYMBEXCEL can infer additional entry points from the VBA code included in the Excel documents, which can be used to trigger the malicious XL4 functionality.

However, the most interesting results emerge from the analysis of the subset of samples that make heavy use of obfuscation and environment-specific variables. For this subset of *symbolic* samples, *XLMMacroDeobfuscator* can only deobfuscate 410 samples. It is interesting to understand why *XLMMacroDeobfuscator* manages to analyze any symbolic samples at all. We find that the analysis can be successful either because the concrete environment values used by the tool happen to satisfy the values expected by the malware sample, or because the sample uses only the value returned by the function `DAY` to deobfuscate the payload.

Tool	All Samples (24,537)			Structurally Distinct Samples (2,265)		
	Concrete	Symbolic	All	Concrete	Symbolic	All
XLMMacroDeobfuscator	11,965	410	12,375	225	363	588
SYMBEXCEL	16,692	7,239	23,931	1,020	1,081	2,101
SYMBEXCELSTATIC	16,644	7,231	23,875	1,014	1,078	2,092
SYMBEXCELCOM	16,660	7,236	23,896	1,018	1,079	2,097

TABLE I: Comparison of the number of samples deobfuscated correctly by SYMBEXCEL and XLMMacroDeobfuscator.

Interestingly, *XLMMacroDeobfuscator* implements a brute-force strategy specifically for the return value of this function. On the other hand, SYMBEXCEL can automatically find the correct environment configuration and deobfuscate 7,239 samples. In this case, the performance difference is primarily due to the symbolic implementation of the function handlers, which allows SYMBEXCEL to accumulate constraints and solve them when executing a symbolic formula to infer the correct environment configuration.

In our experiments, we use a 60-minute timeout, and none of the samples exceed this timeout. Overall, the median analysis time is 12s per sample for SYMBEXCELSTATIC, 32s for SYMBEXCELCOM, and 1s for XLMMacroDeobfuscator.

Triggering Mechanisms. As previously discussed in Section IV-A, malware authors can leverage different ways to start their malicious macros. Figure 5 presents an overview of the triggering functionality we observed in our dataset. The vast majority of documents (15,020 samples) use the *Auto_Open* built-in name to start the execution, while only a few of them leverage instead *Auto_Close* (384) and *Auto_Activate* (6). On the other hand, Figure 5 shows that *VBA* code (9,001 samples) has become a popular triggering mechanism towards the beginning of 2021, clearly in an attempt to evade analysis tools. Moreover, Figure 5 also shows how malicious documents (18 samples) leverage *DCONN* records, which allows Excel to perform a web query and insert new formulas inside a spreadsheet. Despite not being an entry point *per se*, we observed that samples use a combination of *DCONN* and *Auto_Open*. In these cases, a part of the malicious macros is downloaded using the *DCONN* functionality, while the execution is started with *Auto_Open*.

These alternative triggering mechanisms have been used recently—we observed VBA code used as an entry point starting from March 2021—by the malware authors to create samples that are harder to analyze. For instance, at the time of writing, *XLMMacroDeobfuscator* does not support the parsing of VBA entry points. Similarly, the analysis of any sample that uses the *DCONN* functionality is complicated by the fact that part of the Excel 4.0 payload is stored remotely and, thus, cannot be retrieved unless the analysis sandbox is connected to the Internet and the remote server is still reachable.

B. Evasion Techniques

This section discusses the evolution of the evasion techniques used in different waves of Excel 4.0 malicious macros. Initially, we observe such malicious macros separately using hidden macro sheets, control-flow obfuscation, data-flow obfuscation, and sandbox detection checks. However, in later waves, we observe a particular interleaving of sandbox detection checks and data-flow obfuscation that makes the correctness of the deobfuscated code depend directly on the system configuration of the host machine.

Finally, in the latest waves of Excel 4.0 macros, we observe a series of evasion techniques that break the parsing logic of the `xlrd2` parser, as well as the Excel grammar and evaluation logic implemented in *XLMMacroDeobfuscator*, making the samples harder to analyze. In the following paragraphs, we classify the evasion techniques observed in our dataset into seven categories: hidden macro sheets, control-flow obfuscation, data-flow obfuscation, sandbox detection, `xlrd2` parsing confusion, Excel 4.0 macro grammar confusion, and evaluation logic confusion.

Hidden Macro Sheet. Hidden macro sheets are one of the first types of evasion techniques observed in malicious Excel 4.0 macros [28]; our dataset contains samples submitted as early as 2013 that leverage this technique. The visibility of an Excel sheet can be set to *Visible*, *Hidden*, or *Very Hidden*. In particular, while the *Hidden* setting can be changed using the standard Excel User Interface, the *Very Hidden* setting can only be changed using a VBA macro or by manually modifying the binary representation of the macro sheet.

Control-Flow Obfuscation. Multiple functions are used to obfuscate the original control flow of the malicious macro, making the control flow harder to follow by human analysts. First, the `GOTO` and `RUN` functions are used as a trampoline to transfer the execution to target cells, which can reside in different macro sheets. Calls to subroutines—which are linked to either a cell or a defined name—are used along with the `RETURN` function to execute program routines, such as a subsequent stage in a multi-stage macro or a cipher implementation. Finally, the `REGISTER` function is heavily used to register functions from the Windows API with custom names and evade any static deobfuscator attempts to extract strings representing function names, DLL names, URLs, etc.

Data-Flow Obfuscation. We observe several techniques used to obfuscate the data flow of malicious macros. In particular, functions such as `CHAR` or `MID` are heavily used in combination with both basic arithmetic and the `FORMULA.FILL` and `FORMULA` functions to concatenate sets of characters and dynamically generate additional malicious formulas. Moreover, many samples use the *defined names* as a form of temporary storage for strings and intermediate values in general. Finally, recent samples use various types of ciphers to re-arrange, shift, and combine the values in the document to generate the malicious macro.

Sandbox Detection. Malicious Excel 4.0 samples use various strategies to detect a sandbox environment during execution. For example, as described in previous sections, samples use functions such as `GET.WINDOW`, `GET.WORKSPACE`, `GET.DOCUMENT`,

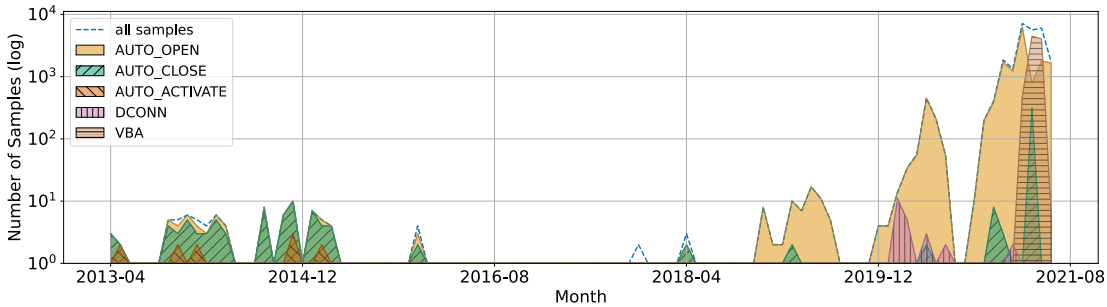

Fig. 5: Distribution of the number of malicious samples using each entry point over time.

`GET.CELL`, etc., to read values from the environment configuration. Moreover, we observed malicious samples that check multiple properties of the host system, such as the existence of a working system clock and a functional file system implementation, or the presence of Alternate Data Streams (ADT), a file attribute specific of NTFS that is commonly leveraged by other types of Windows malware. Some of the samples export and check the contents of the `Excel Macro Security` Windows registry key. Finally, some of the samples use the `Xlcall32:Excel4` API to fork the execution and continue executing the macro in a new process, in an attempt to confuse dynamic analysis techniques that monitor the process execution.

What makes these techniques particularly interesting for SYMBEXCEL is that they evolved from being used as mere environment fingerprinting/evasion mechanisms to playing a key role in the generation of the final malware payload. Instead of simply aborting the execution, these evasion mechanisms provide values that are used to generate the final payload and, therefore, lead to a subtle incorrect deobfuscation of the malware sample. However, SYMBEXCEL can correctly analyze these samples because it treats values related to the execution environment configuration as symbolic.

Static Parsing Confusion. Some samples in our dataset tamper with the Excel file format to cause the static parsing logic to fail. However, at the same time, these samples remain compliant with the parser implemented in Excel, which executes the malicious macro without errors. For example, a common strategy among malware authors is to insert null bytes to alter the representation of the defined names and Shared String Table (SST)—which causes the parsing library to process the malicious file incorrectly (e.g., because of missing defined names).

Excel 4.0 Macro Grammar Confusion. Instead of tampering with the parsing logic, some samples alter the formulas to confuse the XL4 grammar implementation. In fact, the grammar commonly used to parse malicious formulas is a manual approximation of the grammar used by Excel. This approximation allows malware authors to leverage imprecision in the implementation and cause the analysis to fail. For example, Cyrillic and Unicode characters were initially not supported, and parsing a formula with one such character would cause the analysis to abort. Similarly, an incorrect specification of the comparison operator in the grammar resulted in the incorrect parsing of formulas such as `=TRUE=TRUE=TRUE` (a sequence of two comparisons). After noticing this problem, we improved the grammar used by SYMBEXCEL to support such formulas.

Evaluation Logic Confusion. Finally, some samples target the analysis tools at a higher level and introduce functions for which the handlers are either partially implemented or incorrectly implemented. For example, some of the samples observed by the VMware Threat Research Unit [28] make heavy use of arithmetic operations on floating-point numbers to test the correctness of the underlining engine.

C. Malware Evolution

This section discusses how we can leverage the data extracted by SYMBEXCEL to analyze the malware samples contained in our dataset, and classify them into behavioral clusters. This analysis aims to show how malware families evolve, and to identify sub-clusters that represent variations of the same malware family.

Figure 7 in the Appendix presents an overlay of two triangular matrices representing the behavioral (below the diagonal, in the lower triangle) and structural (above the diagonal) similarities between different malware samples. Lighter shades of red indicate lower similarity, while darker shades indicate higher similarity.

In particular, to compute the behavioral similarity clusters, we first pre-process the SRFs, extracting the function names and relevant arguments (e.g., `CALL urlmon URLDownloadToFileA`). We then transform such tokens using the Term-Frequency Inverse-Document-Frequency (TF-IDF) and calculate their *cosine* similarity. Finally, we perform hierarchical agglomerative clustering using the nearest point distance algorithm (i.e., single linkage) with Euclidean distance, and show the resulting hierarchically-clustered heatmap in the lower triangle in Figure 7. With a cutoff threshold of 2, we obtain 40 behavioral clusters (of which 13 are singletons) with an average size of 57 samples. We exclude the clusters with less than 20 samples, and identify nine distinct behavioral clusters with an average size of 239 samples (highlighted in Figure 7) that cover *95%* of the distinct samples in our dataset. Specifically, the distinct behaviors identified by the hierarchical clustering can be recognized by observing red-colored blobs close to the diagonal of the matrix, which are marked with incremental numbers ranging from one to nine.

To compute the structural similarity clusters, we similarly process the list of all functions executed by each sample. We then preserve the ordering from the lower triangle (below the diagonal) and overlay the structural similarities in the upper triangle (above the diagonal). This representation allows us to observe clusters of samples with similar behavior and, at the same time, to compare their structure.

Figure 6 presents a timeline with the number of observations per cluster over time. For the sake of clarity, we only include the six clusters with more than 50 distinct samples in the timeline. Interestingly, this timeline shows that samples belonging to the same cluster

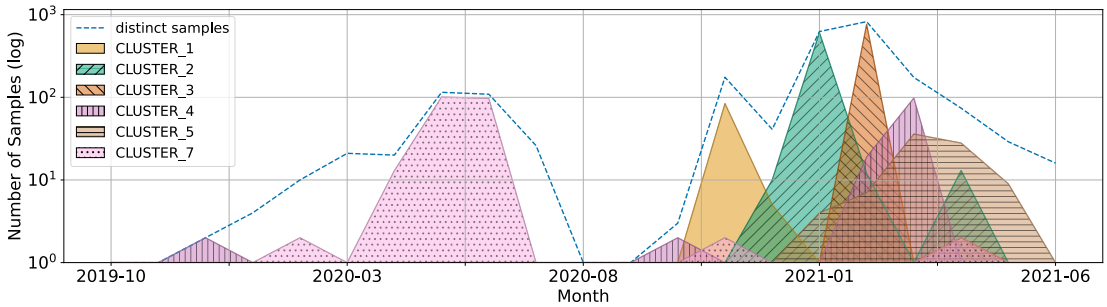

Fig. 6: Timeline of the number of clusters observations over time.

are focused around a given time. This distribution supports our observation that the malware authors will develop new variants as the previous ones are detected and, therefore, are no longer effective.

In the following sections, we present our study of the identified clusters and provide insights into the temporal evolution of the observed behaviors.

Cluster 1. The samples in *Cluster 1* display different behaviors depending on the architecture of the machine where the sample is executed. On a *32-bit* system, the samples download and execute a malicious DLL file. On the other hand, if a *64-bit* system is detected, these samples create on the file-system two *VBScripts* using the `FOPEN` and `FWRITELN` functions, and execute them using *explorer.exe*. The first script file downloads a malicious DLL from a remote server, while the second one executes this DLL using the *rundll32.exe* executable. We observe two variants of the samples belonging to *Cluster 1*. Both variants use three consecutive stages, and heavily use the `SET.NAME` function to update the values of the defined names during the deobfuscation. The first two stages are used to detect sandboxes and to deobfuscate the third stage using values from the environment. The only difference between these two variants is their usage of different functions (e.g., `FORMULA.FILL` instead of `FORMULA`) to write the third-stage payload in the macro sheet.

We first observed *Cluster 1* on November 2020 and—considering their structural and behavioral similarity—it likely represents an evolution of the first cluster described by the VMware Threat Research Unit in October 2020 [67].

Cluster 2. The behavior of the samples in *Cluster 2* is equivalent to that of the samples in *Cluster 1*, as also observed in Figure 7. However, the structure of these samples shows some different characteristics. In particular, samples from *Cluster 2* use a first stage with two nested loops that combine values from both the cells and the environment to generate the second stage. As a result, the correct execution of both the second and third stages depends on the correctness of the environment values. Similarly, we observe three variants of this cluster. The main difference between these variants is the usage of different functions—such as `TRIM`, `CONCATENATE`, `SUM`, `MAX`, `MIN`—during the deobfuscation stage. One of the variants, in particular, uses the `MIN` and `MAX` functions with symbolic arguments in a loop to generate additional possible behaviors.

Cluster 2 was first observed on December 2020 and is likely an evolution of *Cluster 1*, given their behavioral similarity. Moreover, the timeline presented in Figure 6 shows how this cluster was deployed after *Cluster 1* retired, and our observations suggest that these two events are correlated.

Cluster 3. The samples in *Cluster 3* are behaviorally similar to the samples observed in *Cluster 2*, but introduce a two-stage structure with two distinct deobfuscation routines. Also, there is a common second stage that is executed for both *32-bit* and *64-bit* systems. *Cluster 3* was first observed on February 2021, when the number of reports for *Cluster 2* started to decrease. Indeed, after a closer comparison, we observe that *Cluster 3* exhibits a behavior similar to *Cluster 2*, but uses a Javascript (*.js*) file instead of *VBScript* to download and run a DLL.

Cluster 4. The samples in *Cluster 4* were first observed in a simplified form in December 2019. Interestingly, they use two distinct deobfuscation routines and novel evasion techniques in the deobfuscation stage, such as checking the value of the `Excel Macro Security` Windows registry key. The samples of this family evolved around February 2021 and started using more sophisticated evasion techniques, such as the `Xlcall32:Excel4` API to fork the execution. We then observe a wave of samples belonging to this cluster starting on February 2021, at the same time as *Cluster 3*. In fact, we observe that this cluster also uses two distinct routines to deobfuscate its four stages.

Cluster 5. The samples in *Cluster 5* all share a simple structure (on average, 18 formulas) and display a behavior that consists of the download and execution of one or more files. The distinctive trait of this cluster is its heavy usage of the `REGISTER` function. In particular, the majority of these samples use the `REGISTER` function to register a custom function that downloads a file, call the custom function, and finally execute the file with the `EXEC` function. However, some of the samples in this cluster additionally use the `NOW` function to retrieve the current timestamp and generate the target URLs. *Cluster 5* was first observed on January 2021. These samples do not use sophisticated evasion or deobfuscation techniques, likely as a way to blend in with other benign samples.

Cluster 6. The samples in *Cluster 6* are samples that are currently unsupported by SYMBEXCEL (for example, because of imprecisions in the implementation of our functions handlers). That is why the corresponding behavior is shown as a white area in Figure 7. However, SYMBEXCEL can partially execute these samples, which offers a way to measure their structural similarity with other samples in our dataset. In particular, by looking at the structural similarity matrix (above the diagonal in Figure 7), we observe that many of these samples are structurally similar to the samples in *Cluster 1*, indicating that more than 50% of the samples in this

cluster could belong to a new cluster that is a variant of *Cluster 1*.

Cluster 7. We observe two main variants of *Cluster 7*, respectively, in March 2020 and May 2020. Both the variants first check the `Excel Macro Security` settings (a technique already observed in *Cluster 4*). Then, they request the *office-msi-non-security-updates* web page from *docs.microsoft.com*, possibly to mimic a benign behavior. Finally, the samples download and execute a malicious file.

The first variant of *Cluster 7* (observed in March 2020) has a very simple structure (32 formulas) that first deobfuscates a sequence of environment checks and then executes them. The second variant (first observed in May 2020) introduces multiple stages of deobfuscation interleaved with the environment checks. This variant is generally more complex (244 formulas) and also makes heavier use of control-flow obfuscation (`RUN` and `GOTO` functions). Finally, we observe a third variant first used in February 2020 and then deployed again in April 2021. This third variant is less prevalent and uses a series of calls to the `SET.NAME` function during the deobfuscation stage.

Other Clusters. We observe some smaller clusters in the bottom right part of Figure 7, namely *Cluster 8* and *Cluster 9*. The samples in *Cluster 8* are behaviorally very similar to the samples observed in *Cluster 5*. However, these samples do not use the `REGISTER` function, but only download and execute an executable file. Despite being behaviorally similar, the samples from *Cluster 8* do not share a similar structure. Instead, we observe that such samples often use functions that are otherwise rarely used, such as `CEILING.PRECISE`, `RADIANS`, `SUMPRODUCT`, `ACOS`, `SUMXMY2`, etc., possibly as a way to break existing analysis techniques.

Cluster 9 contains samples that heavily use control-flow obfuscation (out of 157 formulas executed, 135 use the `RUN` function). The behavior displayed by these samples is straightforward and consists of the creation of two nested directories using the `Kernel32:CreateDirectoryA` API followed by the download and execution of an executable file.

D. Malware Families

To study the malware families in our dataset, we retrieve and analyze the VT labels of the public samples. In particular, we find Microsoft Defender to be the most reliable for XL4 malware samples.

We first assign a family name to each sample by parsing the Microsoft Defender labels and matching them against a list of verified family names from multiple sources [1], [59], [63]. Quite interestingly, we observe that 15% of the public samples are unlabeled, and 36% have a generic, non-meaningful label (e.g., *TrojanDownloader:O97M/EncDoc*). This clearly shows that XL4 malware remains difficult to analyze, and even authoritative engines are not always able to correctly label these samples, and correlate them with a specific threat actor. Nonetheless, from the remaining 49% of the samples, we identify nine different families— *Donoff*, *Dridex*, *Gozi*, *Hancitor*, *IcedID*, *Mailcab*, *Qbot*, *TrickBot*, and *Zloader*—suggesting that multiple threat actors have abused this infection vector. We select the three most prominent families and present their observations timeline in Figure 8 in the Appendix.

Excel 4.0 malware samples often do not directly infect the host machine, but rather download a secondary infection payload. For this reason, while our behavioral clustering is based on the malicious behaviors observed in the XL4 macro, this does not always correlate with the observed secondary infection payload (i.e., the malware family). As a result, we expect our clusters to loosely match with the VT labels, and we also expect different families to share similar XL4 payloads.

Cluster	IcedID	Qbot	TrickBot	Zloader	O	GEN	UL
1	·	·	·	99%	·	·	1%
2	·	·	·	58%	·	25%	18%
3	·	·	·	89%	·	·	11%
4	·	·	·	29%	·	16%	55%
5	32%	11%	5%	·	·	40%	11%
6	·	3%	1%	25%	2%	57%	13%
7	·	·	·	·	2%	50%	48%
8	25%	27%	2%	·	2%	32%	13%
9	·	·	·	·	·	68%	32%

TABLE II: Distribution of malware families for each behavioral cluster. O: *other*, GEN: *generic*, UL: *unlabeled*.

Nonetheless, we still observe some correlation between the assigned family names and the behavioral clusters presented in Figure 7. We present our results in Table II and distinguish three cases: clusters that belong primarily to one family, clusters that belong to multiple families, and clusters where most samples are unlabeled or have generic labels. In the first case, we observe that clusters 1, 2, and 3 mostly belong to the *Zloader* family. Since the samples in these clusters show a complex behavior, we speculate that they were exclusively deployed by *Zloader* threat actors. In the second case, we observe that clusters 5, 6, and 8 contain samples from different families. We speculate that the simplicity of the behavior observed in such samples justifies its use by multiple and different threat actors. Finally, clusters 4, 7, and 9 mostly contain unlabeled or generically labeled samples. We manually investigate such samples and observe that many are labeled with the generic labels *kryptik* (in cluster 4) and *abracadabra* (in clusters 7 and 9).

Other Variants. The previous discussion focused on the most recent samples in our dataset, but, as shown in Figure 5, our dataset also contains samples submitted during 2013 and 2014. We trace these five structurally distinct samples back to different variants: *Poppy*, *NetSnake*, *Laroux*, and *Yagnuul*. Interestingly enough, *NetSnake* shows a more advanced behavior than some of the newest samples in our dataset: it extracts from a hidden macro sheet column the content of a VBA script, and executes such script. In turn, the VBA script writes a Cabinet (*.cab*) file on disk, unpacks it, and gains persistence by overriding several registry keys to point to the unpacked files.

E. Malicious Behavior Study

This section presents an overview of the different techniques that malware authors use to infect the target system. In Section V-C and Section V-D, we observe that the vast majority of the samples in our datasets are droppers—i.e., a type of malware that downloads and executes an executable file or a DLL [37]. Despite this shared behavior, malware authors use different techniques to download a secondary infection payload from remote hosts. In particular,

the samples in our dataset either use: (1) *powershell* scripts, (2) the `ExtExport.exe` executable from *Internet Explorer*, (3) a Visual Basic script that is executed using `explorer.exe`, or (4) a JavaScript script that is executed using `explorer.exe`.

On the other hand, the most common ways to start the downloaded executable are: (1) via the `explorer.exe` binary, (2) by calling the WinAPI function *ShellExecuteA*, (3) by invoking *rundll32*, or (4) by executing *regsvr32*. Moreover, we noticed a peculiar technique in some samples which combines both the download and the execution in a single step, and utilizes a custom configuration file for `WsatConfig.exe`, a tool normally used to manage transactions between distributed applications [46]. In this configuration file, the malware authors include a reference to a remotely hosted DLL, which is loaded and executed when `WsatConfig.exe` starts. This technique is particularly interesting since it is significantly different from most observed malicious behaviors, and represents an example of DLL injection [31].

During our analysis, we found that some samples reveal different behaviors when executed in different environments. For example, some samples detect whether the architecture is 32-bit or 64-bit, and execute a different payload accordingly. Other samples use a backup strategy when downloading a secondary infection payload: when the connection to a remote server fails, the sample tries to connect to other servers, one after the other, and sometimes even using different methods (e.g., WinAPI, VBScript, JavaScript). Since SYMBEXCEL uses multiple states to keep track of different execution paths, it can effectively capture all of these different behaviors. In our dataset, we observe that samples with multiple behaviors are common: 47% of the structurally distinct samples show two or more behaviors, while 38% show three or more behaviors.

F. IoC Study

While security-relevant formulas can suggest malicious behavior, IoCs are the de-facto standard when responding to security incidents or conducting forensics investigations. For this reason, in this section, we focus on traditional IoCs—i.e., URLs, filenames, domain names, and IP addresses. Our knowledge and control over the SRFs logging format make it straightforward to extract such IoCs from SYMBEXCEL. Similarly, in the case of XLMMacroDeobufuscator, we extract these IoCs by analyzing its execution logs.

The result of this process is presented in Table III. The number of URLs extracted from the output of XLMMacroDeobufuscator and SYMBEXCEL is, respectively, 1,087 and 1,806. We also extract, respectively, 758 and 3,231 filenames, and the most common file extensions are *.vbs*, *.txt*, *.html*, *.reg*, and *.exe*.

We observe that XLMMacroDeobufuscator appears to extracts some IoCs that are not extracted by SYMBEXCEL. The only source of difference between the two tools is their different concretization strategy for timestamps and random numbers. For instance, while SYMBEXCEL uses a fixed value when concretizing a timestamp, XLMMacroDeobufuscator always uses the current timestamp, resulting in different IoC values over time. Considering this difference, we confirmed that SYMBEXCEL's IoCs are a super-set of the IoCs extracted by XLMMacroDeobfuscator.

Finally, we further break down the observed URLs by extracting the unique domain names (451 and 635, respectively) and the unique IP addresses (133 and 215, respectively). After resolving the domain names using historical DNS records from VT, we found that there is no overlap between the domain names and the IPs. We also query the VT intelligence API to verify the reputation of the extracted domains and IP addresses. The results of these queries reveal that 403 out of 635 domains and 212 out of the 215 observed IPs are reported as malicious by at least one antivirus engine. The median detection rate of the public samples is 28 out of 75 engines. However, the median detection rate for the domains and IPs extracted from such samples is 2 out of 90 engines and 5 out of 90 engines, respectively. This result suggests that antivirus engines aggressively label XL4 samples as malicious, but do not properly extract and label the relevant IoCs. To analyze the remaining 232 domains that are classified as benign by VT, we use a combination of two services: urlhaus [2] and AlienVault OTX [6]. The first service flags 141 of such domains as malicious, and the second reports that 39 domains are blocked by Akamai, 22 are not resolved, 12 are automatically generated (DGA), five are generic malicious domains, two are whitelisted, and one is sinkholed. We speculate that the remaining ten domains could be legitimage websites that were likely compromised.

Tool	URLs	Filenames	Domains	IPs
XLMMacroDeobufuscator	1,087	758	451	133
SYMBEXCEL	1,806	3,231	635	215
Total	2,202	3,346	635	215

TABLE III: Breakdown of the IoCs observed in our experiments.

VI. DISCUSSION

Loader. The loader represents a crucial component of our system. Although SYMBEXCEL uses two different strategies—i.e., the static parser and the COM loader—to load Excel files, neither of them is perfect. As discussed in Section V-B, malware authors are continuously devising new ways to break the parsing logic, and sometimes even using the COM functionality is not enough to handle these evasion techniques. This is a known problem in the malware analysis world, and it is not limited to Excel malware. For instance, Nisi et al. [55] recently described the intricacies of the Windows Portable Executable file format, with a focus on the PE loader. The authors found evidence that malware samples can bypass analysis tools by leveraging discrepancies between a tool's loader and the Windows loader, which clearly mimics the evasion techniques we highlighted in this paper. The authors also argue that a *de facto* reference implementation of a loader does not exist, and how different versions of the Windows loader behave in different ways when dealing with the same binary file.

Formula Parsing. Being able to parse the XL4 formulas correctly is crucial to analyze these malicious documents. This process could seem straightforward at first, but the syntactical features of Excel formulas are quite complex. To date, although several attempts have been made to solve this problem [4], [5], [8], we still lack a complete Excel 4.0 formula parser. These attempts have been tailored to *benign* formulas, but, as discussed in Section V-B, malware authors were able to find limitations in the manually reproduced grammar that we use in SYMBEXCEL. A more precise grammar—ideally,

a grammar that matches completely the one implemented in Excel—is therefore needed to handle complex Excel 4.0 malware.

VBA vs. Excel 4. In the latest batches of samples that we analyzed during this research, we noticed that malware authors have started distributing documents containing both Excel 4.0 macros and VBA code in a clear attempt to hinder available tools. The first samples were quite straightforward to analyze and support in SYMBEXCEL, since the VBA code was only used as a "trampoline" to Excel formulas. Unfortunately, this behavior quickly evolved, and the malicious payload started to be laid out over both representations. In particular, we manually analyzed samples where the control flow jumps *back-and-forth* from Excel 4.0 formulas to VBA procedures. Supporting these samples would require including a VBA engine in SYMBEXCEL, such as *ViperMonkey* [38].

Microsoft Policy Change. Microsoft has taken note from the security advisories related to this threat ecosystem and has recently announced that it plans to disable Excel 4.0 Macros by default, although users can still decide to enable this feature. While this change certainly represents a move in the right direction, it is only limited to Microsoft 365 customers and does *not* apply to all Excel users. Moreover, despite this announcement, malware authors are still leveraging Excel 4.0 to infect users and systems, as seen in several samples found in the wild related to the *SquirrelWaffle* campaign [13] (October 2021), to the *WIRTE* campaign [73] (November 2021), to the *Dridex* malware family [54] (December 2021), and to *Emotet* [75], [76] (January 2022).

VII. RELATED WORK

Malware Analysis. In the past few decades, in a joint effort, both industry and academia have extensively studied malware and the surrounding ecosystems. The main focus of this research has been targeted towards malware affecting Microsoft Windows, from studying the behavior of such malware [11], [36], [40], [62], to packing mechanisms [3], [19], [30], [42] and detection of ongoing threats [17], [20], [61], [74]. Moreover, mobile platforms, such as Android, were also studied in the context of malware analysis [16], [60], [71]. Finally, the focus has more recently been targeted at Linux [18], [22] and IoT devices [12], [23], [69], [70], since they became a target for malicious actors. Despite the amount of research on this topic, to the best of our knowledge, we are the first to extensively study Excel 4.0 malware.

Symbolic Execution. Symbolic execution is a powerful analysis technique, and it has been applied to a variety of problems: from automatically revealing the security impact of fuzzer-generated crashes in the Linux kernel [77], to finding vulnerabilities in embedded devices [25] and vetting USB device firmware [29].

In recent years, symbolic execution has also been used in the context of malware analysis [14], [15]. For example, Baldoni et al. [9] developed a tool based on `angr` [65] to automatically analyze a remote access trojan (RAT) and extract its Command and Control (C&C) communication protocol. In a similar vein, Gritti et al. [27] proposed Symbion, a tool based on interleaved execution that is able to study specific malware behaviors. The idea behind interleaved execution is to *concretely* execute a malware sample until the target behavior is reached, and only then switch the analysis to *symbolic* reasoning. This approach, amongst other things, allows Symbion to track the routines responsible for the generation of C&C domains, and to bypass malware evasion techniques.

Symbolic execution has also been applied to the problem of automatically unpacking malware binaries. Ugarte-Pedrero et al. [68] presented Rambo, a tool based on multi-path exploration that can be used to unpack *shifting-decode-frames* packers— i.e., a strategy where a piece of code is unpacked on-demand. Similarly, others [15], [21], [24] have researched the problem of identifying dormant malicious behavior. In particular, Comparetti et al. [21] records specific behaviors observed while dynamically executing a malware sample to identify similar functionality in other programs. Furthermore, Alrawi et al. [7] presented *Forecast*, a tool that combines memory forensics of an infected system with symbolic execution, to predict future malware behaviors.

Sebastio et al. [10], [64] transform symbolic execution traces into a system call dependency graph (SCDG), which summarizes the behavior of the software under analysis, and then uses supervised machine learning to classify the sample into a specific malware family.

The detection and analysis of evasive behavior in malware has been vastly studied [32], [33], [35]. In particular, Kirat et al. [35] automate the differential analysis of evasive malware samples, and present MalGene, a technique for automatically extracting analysis evasion signatures. Moser et al. [52] leverage forced execution to achieve multi-path exploration on malware. One important limitation of this work is that it cannot handle self-modifying code. Symbolic analysis of self-modifying code is a challenging problem that, to the best of our knowledge, has been addressed only in a very restricted scope— i.e., via the concretization of jump targets in x86 binary code [72]. Thanks to the nature of the XL4 grammar and its memory model, we show that this problem becomes solvable in the domain of XL4 macros.

While most of these approaches leverage symbolic execution to drive the execution to specific portions of the code, our approach uses symbolic execution to handle the unknowns about the environment and deobfuscate the malicious code. In this respect, our approach is more similar to approaches, such as Rambo [68], that focus on unpacking. However, our approach is different in its ability to combine metainformation about the structure of the Excel grammar to drive the concretization process and obtain code that can be successfully executed.

VIII. CONCLUSIONS

This paper studies the malware ecosystem surrounding Excel 4.0 macros, a malware distribution vector that recently became popular amongst malicious actors. To achieve this goal, we developed SYMBEXCEL, a tool that can automatically deobfuscate complex Excel 4.0 malware. Our system is based on symbolic execution, a powerful program analysis technique that we use to model interactions between the macro and the environment. This model enables our system to represent the values coming from the environment as symbolic variables and to cope with sandbox detection and formula decryption techniques that are embedded in modern Excel 4.0 malware. Our evaluation highlights how our system represents a clear step forward in the fight against malicious Excel 4.0 macros, since it supports the analysis of a much larger number of samples when compared to state-of-the-art tools.

ACKNOWLEDGMENTS

We want to thank the anonymous reviewers for their valuable feedback. We would also like to thank Sebastiano Mariani and Jason Zhang for their help with the evaluation of our work. Finally, we would like to thank Amirreza Niakanlahiji (also known as DissectMalware) for developing and maintaining XLMMacroDeobfuscator and xlrd2. Among the authors, Christopher Kruegel is VP Security Services at VMware, Inc. Giovanni Vigna is the Sr. Director of Threat Intelligence at VMware. This material is based upon work supported by the National Science Foundation (NSF) under Award No. CNS-1704253, and also by donations from Intel and Activision. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsors.

REFERENCES

[1] abuse.ch. MalwareBazaar: Malware sample exchange. https://bazaar.abuse.ch/, 2022.

[2] abuse.ch. URLhaus: Malware URL exchange. https://urlhaus.abuse.ch/, 2022.

[3] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When Malware is Packin'Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[4] Efthimia Aivaloglou, David Hoepelman, and Felienne Hermans. A grammar for spreadsheet formulas evaluated on two large datasets. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 121–130. IEEE, 2015.

[5] Efthimia Aivaloglou, David Hoepelman, and Felienne Hermans. Parsing excel formulas: A grammar and its application on 4 large datasets. *Journal of Software: Evolution and Process*, 29(12):e1895, 2017.

[6] AlienVault OTX. AlienVault OTX. https://otx.alienvault.com/, 2022.

[7] Omar Alrawi, Moses Ike, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua, Taleb Hirani, Brennan Hill, and Brendan Saltaformaggio. Forecasting Malware Capabilities From Cyber Attack Memory Images. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[8] E. W. Bachtal. Excel Formula Parsing. https://ewbi.blogs.com/develops/2004/12/excel_formula_p.html, 2021.

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. Assisting malware analysis with symbolic execution: A case study. In *International conference on cyber security cryptography and machine learning*, pages 171–188. Springer, 2017.

[10] Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, Jean Quilbeuf, and Stefano Sebastio. Efficient Extraction of Malware Signatures Through System Calls and Symbolic Execution: An Experience Report. *hal-01954483*, 2018.

[11] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A View on Current Malware Behaviors. In *LEET*, 2009.

[12] Calvin Brierley, Jamie Pont, Budi Arief, David J Barnes, and Julio Hernandez-Castro. Persistence in Linux-based IoT malware. In *Nordic Conference on Secure IT Systems*, pages 3–19. Springer, 2020.

[13] Edmund Brumaghin, Mariano Graziano, and Nick Mavis. SQUIRRELWAFFLE Leverages malspam to deliver Qakbot, Cobalt Strike. https://blog.talosintelligence.com/2021/10/squirrelwaffle-emerges.html, 2021.

[14] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Bitscope: Automatically dissecting malicious binaries. Technical report, Citeseer, 2007.

[15] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[16] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2018.

[17] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 122–132, 2012.

[18] J Carrillo-Mondéjar, José Luis Martínez, and Guillermo Suarez-Tangil. Characterizing Linux-based malware: Findings and recent trends. *Future Generation Computer Systems*, 110:267–281, 2020.

[19] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 395–411, 2018.

[20] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P)*, pages 32–46. IEEE, 2005.

[21] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying dormant functionality in malware programs. In *2010 IEEE Symposium on Security and Privacy (S&P)*, pages 61–76. IEEE, 2010.

[22] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux malware. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 161–175. IEEE, 2018.

[23] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The tangled genealogy of IoT malware. In *Annual Computer Security Applications Conference*, pages 1–16, 2020.

[24] Jedidiah R Crandall, Gary Wassermann, Daniela AS De Oliveira, Zhendong Su, S Felix Wu, and Frederic T Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGOPS Operating Systems Review*, 40(5):25–36, 2006.

[25] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[27] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–10. IEEE, 2020.

[28] James Haughom and Stefano Ortolani. Evolution of Excel 4.0 Macro Weaponization. https://www.lastline.com/labsblog/evolution-of-excel-4-0-macro-weaponization, 2020.

[29] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2245–2262, 2017.

[30] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer, 2012.

[31] Moonsu Jang, Hongchul Kim, and Youngtae Yun. Detection of dll inserted by windows malicious code. In *2007 International Conference on Convergence Information Technology (ICCIT 2007)*, pages 1059–1064. IEEE, 2007.

[32] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *2011 IEEE Symposium on Security and Privacy (S&P)*, pages 347–362. IEEE, 2011.

[33] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pages 11–22, 2009.

[34] Sangwoo Kim, Seokmyung Hong, Jaesang Oh, and Heejo Lee. Obfuscated VBA macro detection using machine learning. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 490–501. IEEE, 2018.

[35] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 769–780, 2015.

[36] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *Proceedings of the Network and Distributed System Security Symposium, NDSS. The Internet Society*, 2021.

[37] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitraş. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1118–1129, 2015.

[38] Philippe Lagadec. A VBA parser and emulation engine to analyze malicious macros. https://github.com/decalage2/ViperMonkey.

[39] Philippe Lagadec. oletools - python tools to analyze MS OLE2 files (Structured Storage, Compound File Binary Format) and MS Office documents, for malware analysis, forensics and debugging. https://github.com/decalage2/oletools.

[40] Andrea Lanzi, Monirul I Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *NDSS*, pages 255–264. Citeseer, 2009.

[41] Tal Leibovich and Elad Ciuraru. Identifying Excel 4.0 Macro strains using Anomaly Detection. https://www.linkedin.com/in/tal-leibovich-857bb790/, 2021. DEFCON 29, AI Village.

[42] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *NDSS*, 2020.

[43] Microsoft. Excel functions. https://support.microsoft.com/en-us/office/excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188, 2021.

[44] Microsoft. Excel (.xls) Binary File Format. https://interoperability.blob.core.windows.net/files/MS-XLS/[MS-XLS].pdf, 2021.

[45] Microsoft. Excel (.xlsb) Binary File Format. https://interoperability.blob.core.windows.net/files/MS-XLSB/[MS-XLSB].pdf, 2021.

[46] Microsoft. WS-AtomicTransaction Configuration Utility. https://docs.microsoft.com/en-us/dotnet/framework/wcf/ws-atomictransaction-configuration-utility-wsatconfig-exe, 2021.

[47] Microsoft. XLM + AMSI: New runtime defense against Excel 4.0 macro malware. https://www.microsoft.com/security/blog/2021/03/03/xlm-amsi-new-runtime-defense-against-excel-4-0-macro-malware/, 2021.

[48] Mamoru Mimura and Hiroya Miura. Detecting unseen malicious VBA macros with NLP techniques. *Journal of Information Processing*, 27:555–563, 2019.

[49] Mamoru Mimura and Taro Ohminami. Towards efficient detection of malicious VBA macros with LSI. In *International Workshop on Security*, pages 168–185. Springer, 2019.

[50] Hiroya Miura, Mamoru Mimura, and Hidema Tanaka. Macros finder: Do you remember loveletter? In *International Conference on Information Security Practice and Experience*, pages 3–18. Springer, 2018.

[51] Steve Morgan. Cybercrime To Cost The World $10.5 Trillion Annually By 2025. https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025/, 2021.

[52] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (S&P)*, pages 231–245. IEEE, 2007.

[53] Amirreza Niakanlahiji. XLMMacroDeobfuscator. https://github.com/DissectMalware/XLMMacroDeobfuscator, 2020.

[54] Amirreza Niakanlahiji. https://twitter.com/DissectMalware/status/1466124524282212353, 2021.

[55] Dario Nisi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Lost in the Loader: The Many Faces of the Windows PE File Format. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 177–192, 2021.

[56] Library of Congress (loc.gov). Microsoft Office Excel 97-2003 Binary File Format (.xls, BIFF8). https://www.loc.gov/preservation/digital/formats/fdd/fdd000510.shtml, 2021.

[57] Library of Congress (loc.gov). Microsoft office excel 97-2003 binary file format (.xls, biff8). https://www.loc.gov/preservation/digital/formats/fdd/fdd000510.shtml, 2021.

[58] Philip Treacy. Excel 4.0 Macro Functions Reference. https://www.myonlinetraininghub.com/excel-4-macro-functions/, 2017.

[59] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. Malpedia: a collaborative effort to inventorize the malware landscape. In *Proceedings of the Botconf*, 2017.

[60] Andrea Possemato, Dario Nisi, and Yanick Fratantonio. Preventing and Detecting State Inference Attacks on Android. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS), Virtual, 21st-25th February*, 2021.

[61] Chandrasekar Ravi and R Manoharan. Malware detection using windows API sequence and machine learning. *International Journal of Computer Applications*, 43(17):12–16, 2012.

[62] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[63] Silvia Sebastián and Juan Caballero. Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*, pages 42–53, 2020.

[64] Stefano Sebastio, Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. Optimizing symbolic execution for malware behavior classification. *Computers & Security*, 93:101775, 2020.

[65] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(State of) The Art of War: Offensive Techniques in Binary Analysis. *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 138–157, 2016.

[66] Robert Simmons. Excel 4.0 Macros - The Risk of Hidden Threats in Compound Files. https://blog.reversinglabs.com/blog/excel-4.0-macros, 2020.

[67] Baibhav Singh. Evolution of Excel 4.0 Macro Weaponization, Part 2. https://blogs.vmware.com/networkvirtualization/2020/10/evolution-of-excel-4-0-macro-weaponization-continued.html, 2020.

[68] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Rambo: Run-time packer analysis with multiple branch observation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 186–206. Springer, 2016.

[69] Pierre-Antoine Vervier and Yun Shen. Before toasters rise up: A view into the emerging IoT threat landscape. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 556–576. Springer, 2018.

[70] Huanran Wang, Weizhe Zhang, Hui He, Peng Liu, Daniel Xiapu Luo, Yang Liu, Jiawei Jiang, Yan Li, Xing Zhang, Wenmao Liu, et al. An evolutionary study of IoT malware. *IEEE Internet of Things Journal*, 2021.

[71] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–62. IEEE, 2019.

[72] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 732–744, 2015.

[73] Maher Yamout. WIRTE's campaign in the Middle East 'living off the land' since at least 2019. https://securelist.com/wirtes-campaign-in-the-middle-east-living-off-the-land-since-at-least-2019/105044/, 2021.

[74] Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye. IMDS: Intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1043–1047, 2007.

[75] Jason Zhang. Emotet Is Not Dead (Yet). https://blogs.vmware.com/security/2022/01/emotet-is-not-dead-yet.html, 2022.

[76] Jason Zhang. Emotet Is Not Dead (Yet), Part 2. https://blogs.vmware.com/security/2022/02/emotet-is-not-dead-yet-part-2.html, 2022.

[77] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, 2022. USENIX Association.

APPENDIX

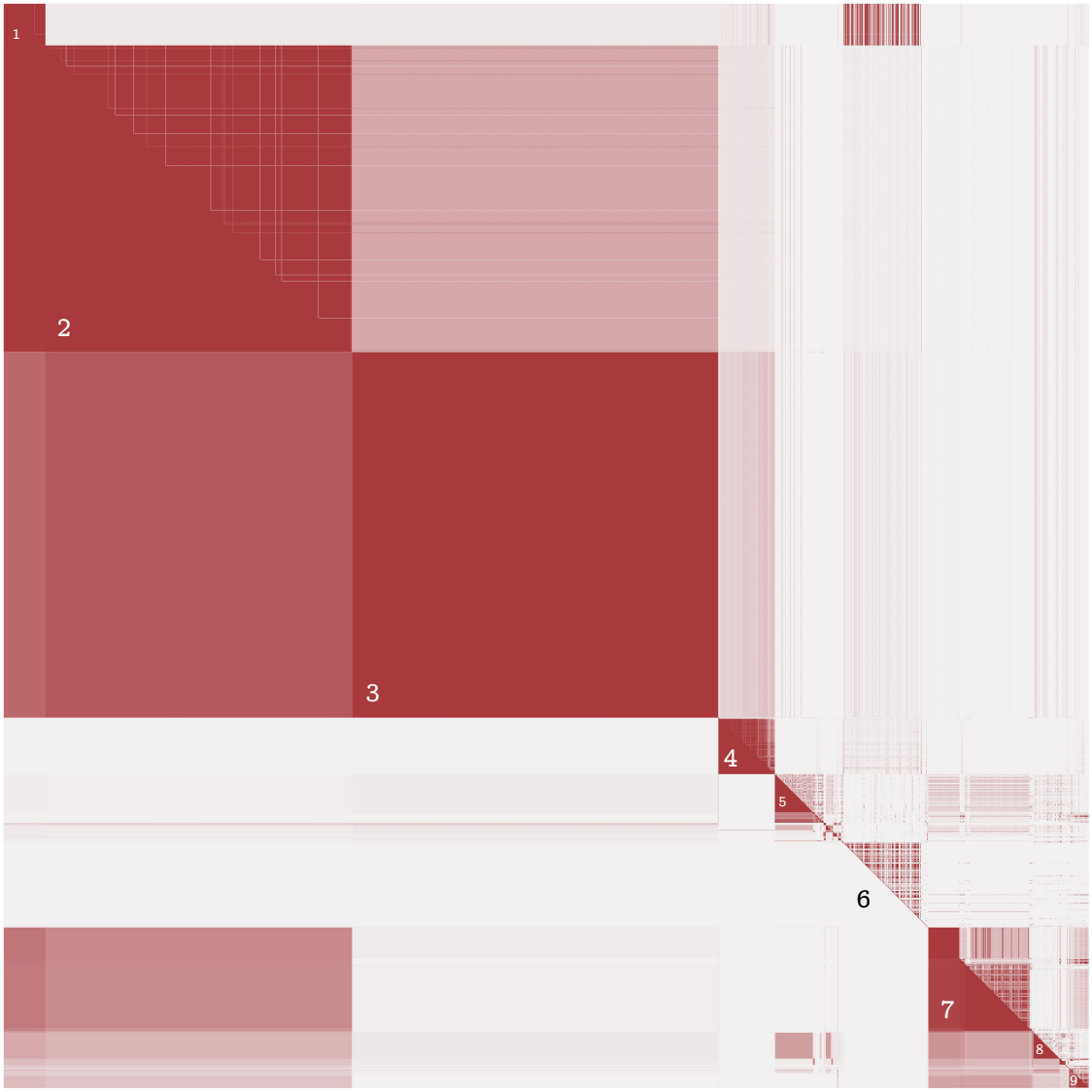

Fig. 7: Heatmap representation of the behavioral (lower triangle) and structural (upper triangle) similarity of the malicious samples.

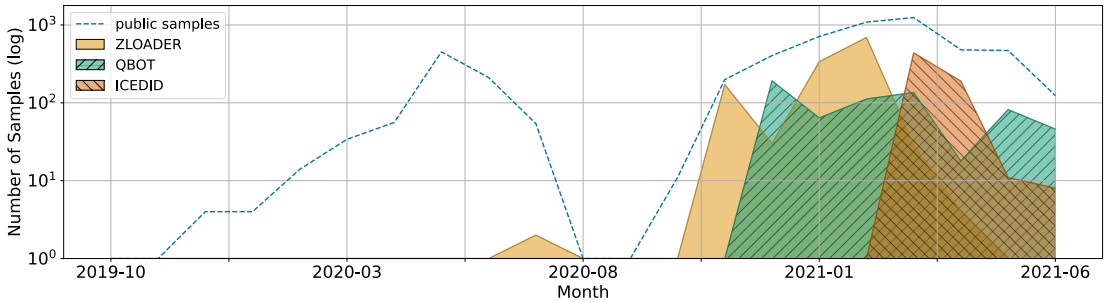

Fig. 8: Timeline of the number of families observations over time.