Forensic Science International: Digital Investigation 44 (2023) 301508

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2023 EU - Selected papers of the Tenth Annual DFRWS Europe Conference

On the prevalence of software supply chain attacks: Empirical study and investigative framework

Anthony Andreoli[*], [1], Anis Lounis [1], Mourad Debbabi, Aiman Hanna

Security Research Centre, Concordia University, Montreal, H3G 1M8, Quebec, Canada

A R T I C L E I N F O

Article history:

Keywords:
Software supply chain attacks
Malicious behaviour
Malware
Binary code static analysis
Dataflow analysis

A B S T R A C T

Software Supply Chain Attacks (SSCAs) typically compromise hosts through trusted but infected software. The intent of this paper is twofold: First, we present an empirical study of the most prominent software supply chain attacks and their characteristics. Second, we propose an investigative framework for identifying, expressing, and evaluating characteristic behaviours of newfound attacks for mitigation and future defense purposes. We hypothesize that these behaviours are statistically malicious, existed in the past, and thus could have been thwarted in modernity through their cementation x-years ago. To measure this, a large scale ground-truth corpus of over 10 million functions is assembled from three file classes: malware, benign, and Windows 10 binaries. An expressive query system is proposed for matching behaviours on top of semantic graphs constructed with data-flow, control-flow and AST-annotations. We leverage conditional probabilities to assess malicious intent by considering the SSCA behaviours matched in the three file classes. Our analysis reveals that the presence of an SSCA behaviour within a binary indicates malware with 86–100% probability. We also annotate each SSCA behaviour with context information that, when applied as a filter over matched dataset samples, is found to boost malicious intent by up to 30%. In addition, we present a novel data-flow metric, *parametric momentum*, which is a powerful gauge of malicious intent that alone matches 12.71% of malware with zero false positives. Finally, we perform a temporal analysis of the SSCA behaviours present in our dataset and discover that they have been available for 13–21 years prior to each attack; conceivably enough time to be identified for mitigating the SSCA instances.

1. Introduction

Software supply chain attacks (SSCAs) have been compromising hosts with steady increase since 2010 (Herr et al., 2020). They typically deliver malicious payload(s) through a tampered end-product of a trusted vendor obtained via established distribution channels, and usually carry a valid digital signature (Ohm et al., 2020). SolarWinds and NotPetya are two notorious cases from a set of over 100 modern attacks of this nature (FIREEYE, 2020; MITRE, 2021). The major concern regarding these attacks is their scale and reach owing to the infection of highly downloaded software in home and enterprise networks. The severity of the risk involved in the software supply chain gave rise to an executive order under the Biden presidency in 2021 that mandates NIST to solicit solutions and best practices from various expert sources (Executive Office of the President, 2021; NIST, 2022).

SSCAs exist along three dimensions: malicious or vulnerable code, open or closed-source, and from an immediate or recursive dependency. The implicit lines of defense within open source software, such as code review, bug reports, and communication between developers are inadequate for eliminating the risk of attack. This work assumes these defenses have failed to catch the compromising injection, and what remains to be detected is the compiled release-binary.

In this research, we consider SSCAs beginning in vendor-side compromise that result in a binary(ies) with injected malicious code; these attacks are considered to have bypassed internal defenses, leaving only the released binaries as a last effort for detection. Seven contemporary and prominent SSCAs that meet the above criteria are selected, namely: Myanmar, League of Legends,

* Corresponding author.
E-mail addresses: anthony.andreoli@concordia.ca (A. Andreoli), anis.lounis@concordia.ca (A. Lounis), mourad.debbabi@concordia.ca (M. Debbabi), aiman.hanna@concordia.ca (A. Hanna).

[1] These authors contributed equally to this work.

https://doi.org/10.1016/j.fsidi.2023.301508

CCleaner, MonPass, NetSarang, Monju, and ShadowHammer. They originate, respectively, from the compromise of: the website belonging to the President of Myanmar, a popular PC video game, a disk management tool, a Mongolian certificate authority, the XShell SSH client, a widely used media player, and ASUS update software. Victims include political entities of Myanmar, gamers in Asia, a Hong-Kong based enterprise, Mongolian hosts, IT and tech companies like Cisco, Sony, and Intel, and the Monju nuclear power plant in Japan.

The primary objective of this work is to thoroughly investigate the malicious nature and the prevalence of behaviours characteristic to the most prominent software supply chain attacks. To this end, a set of research questions are raised:

[RQ1]**Characterization:** What are the code behaviours that characterize the selected SSCAs?
[RQ2]**Maliciousness:** Are the characteristic behaviours exhibited in the SSCAs statistically malicious?
[RQ3]**Contextualization:** Does contextualization of behaviours accentuate malicious intent?
[RQ4]**Prevalence:** Were the SSCA behaviours historically leveraged in known malware?
[RQ5]**Effectiveness:** How effective are detected behaviours in terms of precision and coverage?

To answer such questions, we design and implement an investigative framework that leverages static flow analysis. To assess the prevalence and maliciousness of SSCA behaviors extracted through reverse engineering, we compare them with a very large corpus of benign, malware, and Windows 10 executables. To this end, we formulate the observed SSCA behaviours as queries and search for them in the corpus of binaries represented by semantic graphs. Bayesian inference is finally applied to the query results to asses *malicious intent*, the notable probability that a behaviour exposes malware rather than benign.

We find that if an SSCA behaviour is present, it would indicate malware with 86–100% probability. We also annotate each SSCA behaviour with context information that, when applied as a filter over matched dataset samples, is found to boost malicious intent by up to 30%. In addition, we present a novel data-flow metric, *parametric momentum*, which is a powerful gauge of malicious intent that flags 12.71% of malware with zero false positives. Finally, our temporal analysis of the SSCA behaviours present in our dataset reveals they have been available for 13–21 years prior to each attack, giving ample time for identification and mitigation.

The main contributions of this research are:

1. An investigative approach for binary SSCAs, including behaviour identification, assessment of malicious intent, and a query system that matches behaviours atop semantic graphs.
2. Thorough analysis of the most prominent SSCAs and the derivation of important insights such as their characteristic malicious behaviours, followed by evidence of temporal prevalence from our malware dataset.
3. A set of metrics for assessing malicious intent, including a three-way Bayesian inference over malware, benign, and Windows 10 binaries, contextualization of behaviour for accentuating intent, as well as our novel data-flow metric, *parametric momentum.*

The rest of this paper is organized as follows: Section 2 contains the attack breakdowns, including descriptions of characteristic behaviours for each sample, Section 3 describes the complete investigative approach, Section 4 details the malicious intent of each SSCA behaviour and answers the research questions, Section 5 enumerates contemporary efforts towards understanding and thwarting SSCAs. Finally, Section 6 summarizes our findings and discusses future avenues of research.

2. Background

2.1. Attack threat model

We selected supply chain attacks according to three important criteria. First, the attack is delivered through a supply-chain vendor compromise that enables the attacker to tamper with the software anywhere within the build and release pipeline. Next, the source code of the software must be unavailable, leaving the release binary as the only inspectable artifact. Finally, the sample must be tampered Windows PE software, thus comparable to malware binaries which are highly available for dataset precision verification, and directly intends to compromise through behaviour(s).

2.2. Attack breakdown and behaviour characterization

We dissect each of the studied SSCAs by: A) attack discovery date, B) method of server-side compromise, C) potential victims and targets, D) characteristic behaviour(s) (italicized), and E) possible effects and the final-stage payload of the attack.

ShadowHammer (SH): This supply chain attack, detected in January of 2019, was delivered through an ASUS Live Update, digitally signed, and fetched from ASUS servers (GREAT, 2019; the Robot, 2019). Over 50,000 victims ran the compromised software, however only 600 hosts were directly targeted through hard-coded MAC addresses (Ferguson, 2019). The attack code begins with a concealing technique to reach the `kernel32.dll` module handle via *FS offset* access. This is a set of sequentially dereferenced constant offsets starting with the `FS` processor-register beginning at the thread environment block (TEB, offset 0x18), and eventually reaches the sought-after module (Hyvärinen, 2019; Malvica; Chappell, 2016). The code subsequently digs for the function address of `GetProcAddress` and `LoadLibraryExW` through its *export table*. The table holds the exported functions sought-after within a module, reachable by a sequence of dereferenced constant offsets starting from the module's PE signature (Damaye, 2017). Next, 1–4 byte *sequential constants* (back-to-back stored constants) were written to the stack that when concatenated in memory hold module names like “kernel32”. The next-stage payload is fetched via a network call, loaded into executable memory, and executed. The effects of this payload are unavailable due to the broken malware URL (Hyvärinen, 2019).

CCleaner (CC): In this supply chain attack, Piriform's build servers were compromised with a re-compiled C runtime library (CRT) that replaced the `get_tls_callback` function with malicious code in the digitally signed CCleaner build. The attack was discovered on September 12th, 2017 (Vlček, 2018; Brumaghin et al., 2017b) and targeted large IT and technology companies like Cisco, Sony, and Intel (Brumaghin et al., 2017a). The attack starts by decoding data-section memory via *XOR-in-a-loop* – the looped decryption of a buffer through the `XOR` operation. Heap *executable memory* is created via `HeapAlloc`, filled with the decoded data, and launched. The attack eventually runs ShadowPad, a cyber-attack platform with modular plugin capability.

NetSarang (NS): The network management tool XShell contained the tampered dependency `nssock2.dll`. In this SSCA, perpetrator-access to download servers was suspected, followed by either modification of the build system, or complete replacement of the installer using a valid signature, and was detected on August 4th of 2017. The software affects global networks, servers, and system administrators, with a confirmed, activated payload in a Hong Kong-based company (Beltov, 2017; GREAT, 2017). The attack leverages the CRT-based `initterm` function that runs prior to `main` and normally executes state-initializing code referenced by a vector of pointers; this vector is manipulated to include the address of a malicious function (WeiRanLab, 2018). Its code begins by allocating *executable memory*, filling it with a buffer decrypted with *XOR-in-a-loop*, and then executing it, eventually allowing the C&C to execute arbitrary code (Kaspersky, 2017).

League of Legends (LoL): Riot Games, the company that developed the game, was infected when trojan(s) in the computers and patch servers of Garena (its partner) led to compromise of the game's installer and updater (Ryansecurity, 2015). The supply chain attack, discovered in January of 2015, mainly targets gamers in Asia, with 82% of affected users from Taiwan (Trend Micro, 2015; GameRant, 2015). The initial attack code resides in the DLL entry-point imported by the main executable. The DLL reads itself as a file to access data that is run through *XOR-in-a-loop* and yields a file-path to a `dat` file whose payload is decrypted via *XOR-in-a-loop*, stored in *executable memory*, and executed. The attack proceeds to load several functions from the *export table* into the data section, one of which may be called to complete the attack's installation. This trojan finally deploys PlugX, a remote access tool allowing full control, malicious execution, and exfiltration of data.

MonPass (MP): This Mongolian certificate authority experienced a breach on its public-facing web servers that led to an unsigned malicious installer that was discovered on March 24th, 2021; the victims were mostly Mongolian users (Camastra et al., 2021). The attack begins with *conditional process termination*, which is a system information check (e.g., low RAM) that leads to a process-terminating function (e.g., `exit`). This is an anti-analysis technique possibly indicating malicious intent (Brand et al., 2010). The code then performs *reflective loading* by fetching several functions through `LoadLibrary` and `GetProcAddress`. Subsequently, an image is requested from an HTTP *url string*. Its pixels are read at 4-byte intervals into *executable memory*, and passed to `CreateThread` for threaded execution. The final payload deploys and leverages Cobalt Strike (penetration testing suit) to accomplish undisclosed attack goals.

Myanmar (MY): The website representing the President of Myanmar was compromised with a hijacked version of the Myanmar Unicode font package to include a malicious loader in `Acrobat.dll` (Cimpanu, 2021). The compromise was detected on June 2nd, 2021, and potentially targets Myanmar-based political entities based on possible attribution to the Mustang Panda APT and an attack on the same website in 2014 (CyberSecurityHelp, 2021; Falcone, 2015). The attack begins in an exported DLL-function. The functions `Create/OpenEventA` are leveraged to ensure a single instance of the process is running. A *module filename sink* is performed, where the parent-exe path is fetched via `GetModuleFilenameW` and is used to copy itself and the malicious DLL into a Windows font folder by sinking into `CopyFileW`. The registry, modified via a *registry string*, is set to run this executable and accompanying malicious DLL at startup through the “reg add” shell command. Code from the data section is decrypted and loaded into *executable memory* with protections set by `VirtualProtect`, and finally passed to `CreateThread` for parallel execution. The next-stage payload runs the Cobalt Strike framework.

Monju (MJ): The GOM media player update server was compromised to redirect clients to a second compromised webserver, where the legitimate update and malicious counterpart are fetched and installed. The attack was first detected on January 2nd, 2014, executing on a machine within the Monju reactor facility that contained staff training documents and over 40,000 emails — though official target(s) are unknown (Context Threat Intelligence, 2014). The attack proceeds with *XOR-in-a-loop sinks*, where several module strings decrypted through *XOR-in-a-loop* sink into `LoadLibrary` (i.e., “kernel32.dll”, “shell32.dll”). Returned handles are used to traverse their corresponding *export table* until a set of target functions are found. Several strings are decrypted that hold the names of files to be copied to the `temp` directory through *module filename sinks* of `CopyFileA` and `CreateFileA`. A file included in the software package is decrypted via *XOR-in-a-loop*, stored into another file, and communicated to the registry using a decrypted registry-path and its `InprocServer32` key. Finally, the `explorer.exe` process is reset. A COM-hijacking technique is employed where the default binary object set in `InprocServ32` that responds to COM requests for a given CLSID (windows registry Class ID) (López, 2020) is updated to a malicious binary. When a Windows system process references this COM object, the malicious binary is launched. The final executed payload is known as Ghost RAT, frequently leveraged by Chinese actors, and is capable of exfiltrating system-specific details, and deploying additional malware.

3. Methodology

Our investigative approach operates in four steps and is depicted in Fig. 1. First, SSCA samples are selected through a vetting process and their behaviours identified. Second, these are manually translated to data-flow queries, validated in the *query validation cycle*, and stored in the *query repository*. Third, the collected dataset binaries are processed into semantic graphs and matched against the validated SSCA queries in the *behaviour search engine*. Finally, the resulting matches are evaluated through *statistical inference* to assess maliciousness. In what follows, we detail the inner workings of each phase of our investigative approach.

3.1. SSCA binary retrieval and behaviour identification

The Atlantic Council's SSCA dataset (Herr et al., 2020) provided us with the starting point for looking up binaries that satisfy our threat model. From over 100 SSCA incidents, we select seven attack samples provided by VirusTotal. The entry attack-function in each sample is found using post-mortem reports. Each function reachable from this point is manually inspected and summarized by a list of its behaviours. Criteria is applied to select behaviours that are most frequent between samples, and further intuitively ranked by their perceived utility to malware rather than a legitimate application. An example of this is a buffer decrypted with *XOR-in-a-loop* that sinks into a Win32 API function.

3.2. Binary dataset

Malware samples are provided by a private industry partner as a listing of {MD5, CARO name} (Microsoft, 2022) from which a subset of Windows PEs are selected through the following diversification process. A mapping is created from each malware type to its family-variant groupings, where fv_N represents the number of samples in a family-variant grouping:

$$Malware_{type} \rightarrow [fv_1, fv_2, \ldots, fv_N]$$

The 50th percentile of the codomain's frequencies are retained

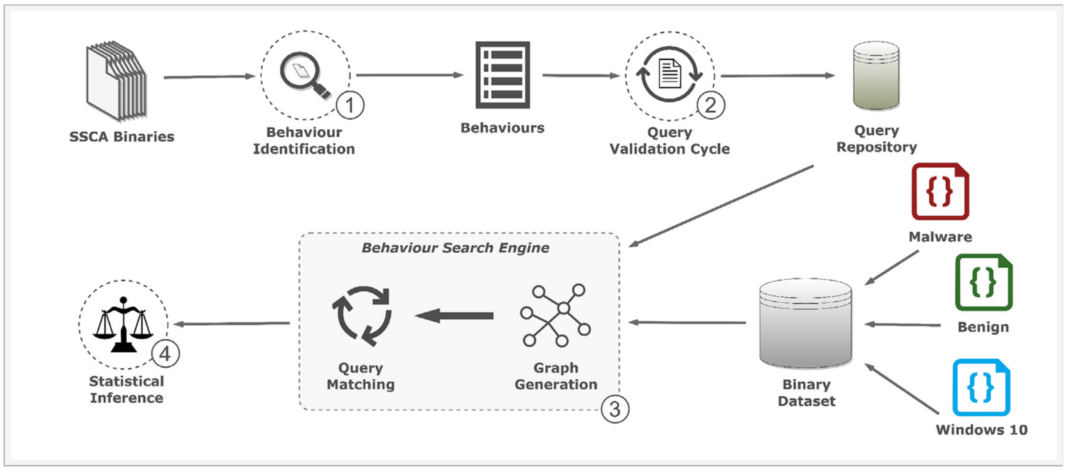


Fig. 1. Approach architecture diagram.

from each grouping to balance representation, and the final selection is randomly reduced to roughly 22,000 samples, with 19,344 EXEs and 2819 DLLs. Benign samples are mined through software sharing websites[2] and are subsequently unpacked, or decompressed. We retain DLLs and EXEs successfully scanned with *Windows Security Defender*. The benign dataset contains a total of 34,419 binaries with 9880 EXEs and 24,539 DLLs. Windows 10 PE files come from a fresh OS installation with 17,851 binaries with 2712 EXEs and 15,139 DLLs. The final dataset excludes.NET binaries.

3.3. Behaviour search engine

The behaviour search engine takes a binary from the dataset, generates a semantic graph for each function, runs all queries from the *query repository* over each, and outputs all matching instances.

Function Selection: The set of analyzed functions in a particular binary are those that mimic the execution-contexts of the inspected SSCAs; those recursively reachable from either the PE entry-point (entry), an exported function (export), or the `initterm` vector (initterm, see Section 2.2).

Function Contextualization: We tag each function with context-metadata, a triple composed of: (a) the recursive-parent entry point of the function, which is one or more of (*entry*, *initterm*, *export*), (b) its binary type (EXE, DLL), and (c) the function's minimum call-graph depth from each of its entry point(s). We select the minimum depth with the intuition that an attacker would likely want to execute its malicious payload closest to the entry point of a binary.

Graph Generation: The semantic graphs generated include data-flow, control-flow, and AST-annotations using Ghidra's decompilation p-code. The set of p-code operations present within of a function is defined as *Ops*. For any operation $op \in Ops$, $In(op)$ yields its input variable(s), and $Out(op)$, the computed output value. Edges are labelled to retain the semantic relationship between p-code operations and their operands; for brevity the labels are not covered in this paper. Data-flow edges, *DFE*, are then defined as:

$$DFE = \bigcup_{op \,\in\, Ops} (\{\, (in,\ op) : in \in In(op) \,\} \cup \{\, (op, out) : out = Out(op) \,\})$$

The operation(s) executed immediately after op are given by $next(op)$. The control-flow edges, *CFE*, are defined as:

$$CFE = \bigcup_{op_u \,\in\, Ops} \{\, (op_u,\ op_v) : op_v \in next(op_u) \,\}$$

Finally, the scope s in which an operation is executed, either the condition (predicate) or body of a control-construct, is provided by $scope(op)$. The scope's parent construct c (while, if, else, for, do, root) is marked by $cstr(op)$. An operation or control-construct in the immediate body of a function is considered within the *root* scope. Note that a construct c may itself be located within a scope; either a construct's body, or the *root*, and is retrieved by $body(c)$ (e.g., a while-loop in the body of an `if`). The AST-annotation edges, *ASTE*, are defined as:

$$ASTE = \bigcup_{op \,\in\, Ops} \{\, e \in \{\, (op,\ ct),\ (ct,\ c),\ (c,\ b) \,\} : s = scope(op) \wedge c = cstr(op) \wedge b = body(c) \,\}$$

The final graph, composed of all three edge sets, is defined as $G = DFE \cup CFE \cup ASTE$.

Query Matching: In our queries, the unit of matching is a data-flow path, which is formally a *directed walk* over the semantic graph. We introduce a declarative approach to capture desired paths using *data-flow expressions*. An SSCA behaviour, for example, is described in terms of one or more expressions. We define data-flow expressions in Backus Normal Form (BNF) in Fig. 2, and its corresponding semantics in Fig. 3.

These expressions are composed of *clauses* that allow defining a data-flow path *within* a certain AST-scope (e.g., a loop-body, root, etc …), starting *from* a desired set of source nodes, leading *to* a desired set of sink nodes, optionally through any number of data-flow edges via a *wildcard*. These clauses capture the semantics of

$$\begin{aligned} scope &::= \text{“}within\text{”}\ source \\ source &::= \text{“}from\text{”}\ path \\ path &::= \text{“}wildcard\text{”}\ sink \mid sink \\ sink &::= \text{“}to\text{”}\ sink \mid \text{“}to\text{”}\ path \mid \text{“}to\text{”} \end{aligned}$$

Fig. 2. Data-flow expression syntax.

[2] download.cnet.com, portableapps.com, win.softpedia.com.

$$
\begin{aligned}
within(sp) &= \bigcup_{c\,\in\,cs(sp)} \{\{\ op \in Ops\ :\ c \in rc(op)\ \}\} \\
cs(sp) &= \bigcup_{op\,\in\,Ops} \{\ c\ :\ c = cstr(op)\ \wedge\ sp(c)\ \} \\
from(np) &= \bigcup_{n\,\in\,DFE} \{\ n\ :\ np(n)\ \} \\
to(np,\ ep) &= \bigcup_{(u,v)\,\in\,DFE} \{\ (u,v)\ :\ np(v) \wedge ep(u,v)\ \}
\end{aligned}
$$

Fig. 3. Data-flow expression clause set-notation.

a behaviour through a scope, sources and sinks, and the data-flow paths between them. We instantiate clauses with custom, behaviour-related predicates to support capturing data-flow semantics along the expression. We define these predicates as the scope-predicate *sp*, the node-predicate *np*, and the edge predicate *ep* acting on data-flow labels. The function *rc* applied on operation *op* provides the set of its recursive parent control-constructs, including the *root* (as defined in Section 3).

An example data-flow expression matching a simplified variant of *XOR-in-a-loop* over its containing graph can be found in Fig. 5. The *within* clause indicates the data-flow path must be exhibited in the body of a loop. Next, a *from* clause requires a LOAD operation at the source of the path. The *wildcard* that follows implies traversing any edge(s) until an XOR operation is reached through any operand, as dictated by the *to* clause. The remaining clauses denote a traversal until the STORE operation is reached through a source edge.

Algorithm 1 matches data-flow expressions over semantic graphs by computing partial matches between successive clauses, and merging them to form the final matched paths. Clauses are evaluated on the graph according to the semantics in Fig. 3 by the `eval` function. The mandatory *within* clause is evaluated with `evalAndMask` to yield a set of scoped subgraphs. A subgraph corresponds to the original graph masked with operations from an individual AST-scope satisfying *sp*. Initially, the *from* clause is evaluated to select the starting source node(s). Subpaths are constructed at each iteration, where their sink nodes become the new `sources` (line 22).

When a *to* clause is evaluated (lines 11–14), it yields single-edged paths from the current `sources`. When encountered, a *wildcard* clause indicates the capture of all paths of arbitrary length from the current `sources` to the edges of the subsequent *to* clause. An `allPaths` algorithm is applied to this end, where only simple paths are considered. Partial matches are merged to produce the set of complete matches satisfying the entire expression (line 24). The partial matches are grouped in a list denoted $[s_1, s_2, \ldots, s_N]$, where s_N is a set of subpaths. The path $p_i \in s_m$ merges with a path $p_j \in s_{m+1}$ if $p_i.sink = p_j.source$ and produces a new path p_k by concatenation.

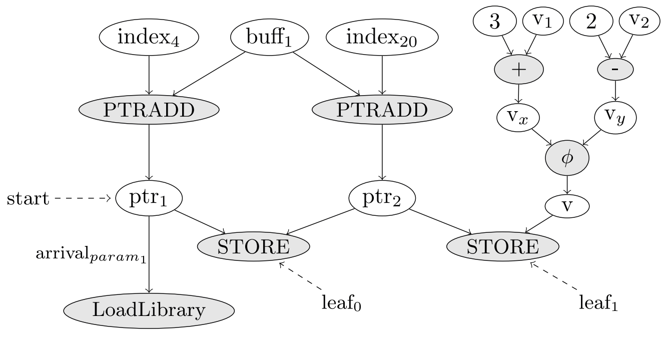


Fig. 4. Parametric Momentum graph.

The result is the aggregate merge of all matches across the subgraphs.

Algorithm 1: Data-flow Expression Matching

```
1  Function match(dfexpr: Queue, graph: Graph):
2      within ← dfexpr.pollClause();
3      maskedGraphs ← evalAndMask(within, graph);
4      matches ← ∅;
5      foreach g ∈ maskedGraphs do
6          expr ← dfexpr.copy();
7          from ← expr.pollClause();
8          sources ← eval(from, g);
9          partialMatches ← List();
10         repeat
11             if expr.peekClause() == to then
12                 to ← expr.pollClause();
13                 sinkEdges ← eval(to, g);
14                 subpaths ←
                     { e ∈ sinkEdges | e.origin ∈ sources };
15             else // wildcard
16                 expr.pollClause();
17                 to ← expr.pollClause();
18                 sinkEdges ← eval(to, g);
19                 subpaths ← allPaths(g, sources,
                     sinkEdges);
20             end
21             partialMatches ← partialMatches + subpaths;
22             sources ← { path.sink | path ∈ subpaths };
23         until expr.isEmpty();
24         matches = matches ∪ merge(partialMatches);
25     end
26     return matches
```

3.4. Query validation cycle

The query validation cycle ensures the validity of a translation from a behaviour to its query. A behaviour is translated into a *query definition* by expressing its code-operations through the query syntax. Artificial code samples with more complex data-flow variants are written to generalize the possible forms encountered. The example translation in Fig. 5 shows data-flow components of the original *XOR-in-a-loop* with variant portions (right-shift) in red. When a query fails to detect the behaviour samples, the matching is debugged by inspecting unmatched subpath(s), and adjusting the query accordingly. This cyclical approach is applied until all instances are matched. The validated queries are stored in the *query repository*.

3.5. Parametric Momentum

This metric, implemented as a query, is designed to summarize the operations applied to a variable through data-flow until it reaches a call as an argument. Parametric Momentum (PM) is inspired by League of Legends, which performed a set of operations on a buffer prior to sinking into `LoadLibrary` – a function generally taking a literal string as input and thus not manipulated by any operation. It is “parametric” as it sinks into a *parameter*, and contains “momentum” as it accumulates.

If the target argument is a primitive type, a simple backwards traversal starting from the argument is collected as a data-flow path. An algorithm then computes the corresponding PM defined as $\sum weight(operation_i)$, where the i^{th} operation encountered in the path towards the call is associated with a weight, adjusted to diminish or accentuate its presence. In this work, we apply a weight of 1 to all operations except non-mutative ones, which are set to zero. If the target parameter is a pointer, a taint graph must be assembled prior to the computation of PM. The system works backwards and forwards between `STORE` operations to collect a coarse-grained backwards taint graph. A sample graph depicting this case over `LoadLibrary` is illustrated in Fig. 4 with a description of the PM computation steps below.

Computation begins over the arrival edge and start node, which connect directly to the sink function. A recursive, backwards traversal with this initializing pair sums each encountered operation according to its weight. A phi-definition (ϕ) operation (multiple definitions of a single variable) has a PM of the *average* of its parent nodes. In all other cases, the PM of a node is the *sum* of its parents' PM. As seen in Fig. 4, this backwards traversal must be repeated for each `leaf` node in addition to the initial start node $\mathtt{ptr}_1$. Leaves are targeted as they represent `STORE` operations that by nature have no output value; the operation stores data from an input variable into a pointer (mimicking $*\mathtt{ptr} = \mathtt{v}$). The final PM value, then, is the sum of each backward-computed PM.

3.6. Statistical inference: behaviour maliciousness

We quantify a behaviour's malicious intent, or *maliciousness*, by its probability of being found in malware binaries, relative to benign and Windows 10. Thus, conditional probabilities are calculated using Bayes' Theorem and the Law of Total Probability. For a given behaviour captured by query q_x, the probability of its appearance indicating a certain class of file $c \in C$, where C = {malware, benign, Windows 10} is:

$$P(c \mid q_x) = \frac{P(q_x \mid c)*P(c)}{P(q_x)} \quad \text{and} \quad P(q_x) = \sum_{c}^{C} P(c)*P(q_x \mid c)$$

A behaviour is considered malicious if $P(Malware \mid q_x) \gg P(Benign \mid q_x) + P(Windows\ 10 \mid q_x)$, and can thus be marked as a conclusive candidate for detecting the specific attack.

4. Results and discussion

In this section, we report several key insights regarding the malicious intent of SSCA behaviours, the important contexts applied to accentuate intent, the subtleties in behaviour across attacks, the application of the novel metric *parametric momentum*, and answers to the research questions.

4.1. Evaluation of behaviour maliciousness

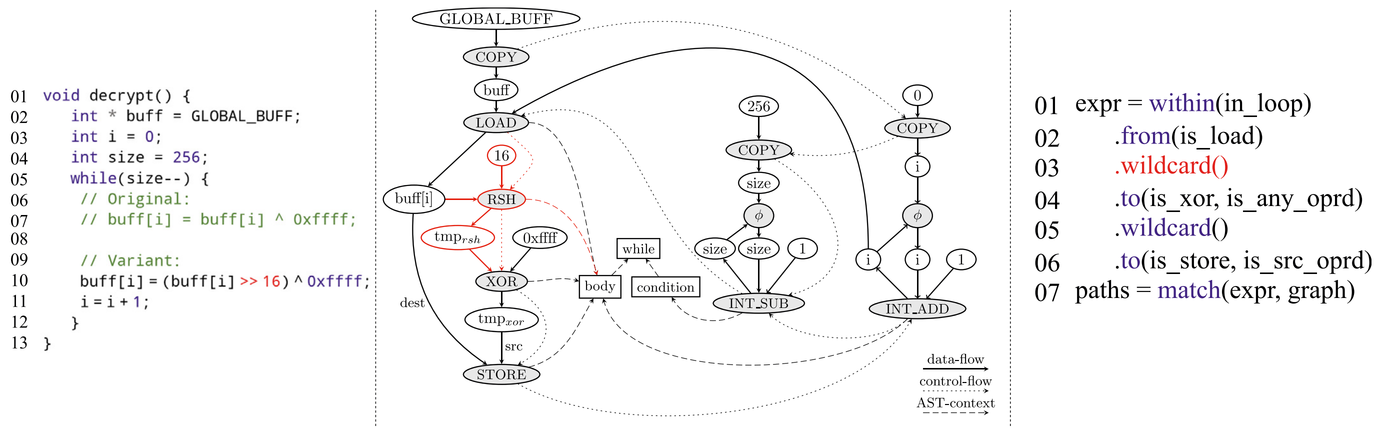
This subsection answers two research questions. The first, **RQ1 (characterization)** is responded with the high-level breakdown of behaviours seen in Section 2, with additional subtle commentary here. Furthermore, Table 1 maps each behaviour to its respective attack for reference. The second question, **RQ2 (maliciousness)** is answered by analyzing behaviour-matching results over the binary dataset.

We discuss the SSCA behaviours and their maliciousness in the boldface-delineated paragraphs that follow. The objective regarding evaluation of behaviours is to leverage optimal context information for each behaviour — through the application of a context filter — to detect the largest number of malware samples while providing maximal maliciousness, a process similar to using a ROC curve for finding the optimal threshold in a classifier. Applying a context filter, for example *(exe, !export, [1, 3])*, retains matches within EXE binaries, reachable through any entry point excluding exports, and at a depth between 1 and 3 calls from the entry point; an attribute marked $ is ignored during filtering.

The results are summarized in Table 1, where the *applied filtering* column indicates the context filter over the dataset results, keeping only those matching the context of the target SSCA (Section 3). The three columns that follow indicate both the conditional probability of the behaviour, and the number of detected samples for each of the three binary file classes. A value of $P(Mawlware \mid q) = 100\%$ indicates absolute malicious intent, while a value of 90% and above is considered "highly malicious" from the dataset's perspective. An SSCA behaviour not found in any dataset is considered anomalous, and is unlikely to have been found in the past. The final three columns indicate the behaviour's attack of origin, the number of detected samples with a compilation timestamp prior to its attack date, and the median difference in years between date of attack and the matched samples. The compilation timestamp is the best available approximation for dating the samples.

Some conclusions made in this section, such as the highly malicious nature of export-table behaviour within an *exe*, may appear trivial. Nevertheless, the framework and results produced are necessary to quantitatively prove such a triviality.

FS offsets: ShadowHammer exhibits a specific sequence of FS-register offset dereferences that reach the loaded-modules of the



```
void decrypt() {
  int * buff = GLOBAL_BUFF;
  int i = 0;
  int size = 256;
  while(size--) {
   // Original:
   // buff[i] = buff[i] ^ 0xffff;

   // Variant:
   buff[i] = (buff[i] >> 16) ^ 0xffff;
   i = i + 1;
  }
}
```

```
expr = within(in_loop)
    .from(is_load)
    .wildcard()
    .to(is_xor, is_any_oprd)
    .wildcard()
    .to(is_store, is_src_oprd)
paths = match(expr, graph)
```

Fig. 5. A. Attack code (left), B. Extracted semantic graph (center), c. Behaviour query (right).

Table 1
Bayesian inference over query results.

Query description	Applied filtering	$P_{mal \mid q}$		$P_{bgn \mid q}$		$P_{w10 \mid q}$		Prior evidence		
	(type, context, depth)	%	#	%	#	%	#	SSCA	# prior	Δyrs.
FS offsets	(exe, $, $)	100.0	125	0.0	0	0.0	0	ShadowHammer	112	5.66
Export Table Offsets	(exe, $, [0, 3])	100.0	428	0.0	0	0.0	0	ShadowHammer	388	8.14
	(dll, entry, [0, 4])	98.25	56	1.75	1	0.0	0	League of Legends	42	5.32
	(exe, $, [0, 2])	100.0	341	0.0	0	0.0	0	Monju	257	4.5
Export Table Looping Predicate	(exe, $, $)	100.0	304	0.0	0	0.0	0	ShadowHammer	270	7.77
								Monju	195	4.26
	($, entry, $)	99.41	337	0.59	2	0.0	0	League of Legends	237	4.62
Export Table Byte-Comparison								ShadowHammer	170	9.1
	($, !export, $)	100.0	196	0.0	0	0.0	0	Monju	138	5.38
								League of Legends	149	5.8
Single XOR-in-a-loop	($, !export, [0, 1])	93.08	875	6.9	65	0.0	0	NetSarang	791	9.25
	($, entry, [0, 2])	94.67	1597	5.28	89	0.06	1	League of Legends	1203	6.22
	(exe, entry, [0, 3])	95.98	1933	3.87	78	0.15	3	CCleaner	1678	7.37
Multiple XOR-in-a-loop	(exe, $, [0, 2])	96.10	370	1.82	7	2.08	8	Monju	223	4.79
XOR-in-a-loop sinks$_{q1}$	($, !export, [0, 1])	100.0	41	0.0	0	0.0	0	Monju	32	4.94
XOR-in-a-loop sinks$_{any}$	($, !export, [0, 2])	96.09	246	3.90	10	0.0	0	Monju	158	4.79
Registry-string reference	($, $, $)	100.0	11	0.0	0	0.0	0	Myanmar	11	8.43
Sequential Constants	(exe, $, [0, 3])	97.74	216	2.26	5	0.0	0	ShadowHammer	186	4.95
Module Filename Sink$_{q1}$	($, $, [0, 1])	97.01	65	3.0	2	0.0	0	Myanmar	63	11.99
Module Filename Sink$_{q2}$	(exe, $, [0, 1])	94.71	269	5.28	15	0.0	0	Monju	239	6.65
Reflective Loading	($, $, $)	99.53	212	0.47	1	0.0	0	MonPass	200	6.33
URL String	($, entry, [1, 2])	86.57	187	13.42	29	0	0	MonPass	187	12.26
Conditional Process Termination	($, entry, [0, 2])	91.29	818	5.35	48	3.34	30	MonPass	802	10.44
	($, $, [0, 1])	85.6	535	11.04	69	3.36	21	Myanmar	525	10.86
PM(LoadLibraryA)$_{p1 \geq 21.0}$	($, entry, $)	95.69	222	4.31	10	0.0	0	League of Legends	173	5.4
PM(GetProcAddress)$_{p2 \geq 10.25}$	($, entry, $)	81.65	654	17.60	141	0.75	6	League of Legends	500	4.47
PM(VirtualAlloc)$_{p1 \geq 24.125}$	($, $, [0, 4])	92.16	47	5.88	3	1.96	1	League of Legends	20	2.45
PM(VirtualAlloc)$_{p1 \geq 24.125 \wedge p2 \geq 21.125}$	($, $, [0, 4])	93.18	41	4.54	2	2.27	1	League of Legends	18	2.45
VirtualAlloc$_{explicit}$	($, !export, [1, 1])	100.0	348	0.0	0	0.0	0	NetSarang	274	7.31
	($, !export, [0, 3])	95.13	1133	4.53	54	0.33	4	League of Legends	798	5.33
VirtualProtect$_{explicit}$	($, $, [1, 1])	97.97	145	0.0	0	2.02	3	Myanmar	136	7.61
VirtualProtect$_{reflective}$	($, $, $)	100.0	49	0.0	0	0.0	0	MonPass	49	2.72

executable. Querying for this behaviour involves latching onto the initial FS-read instruction and following its associated data-flow; it reveals malware 91.25% of the time. Under the context of *exe-only*, it is an absolute indicator of malicious intent (100%). This contextualization is effective as Windows 10 DLLs exhibit this behaviour within export-functions.

Export Table [ET] Offsets: Reaching the ET is captured by a set of consecutive offset dereferences; an important behaviour found in ShadowHammer, League of Legends, and Monju because of its concealed access to function pointers within a loaded module. ShadowHammer suggests absolute malicious intent with context *(exe, $, [0,3])*. Monju, with a subset context of ShadowHammer, achieves absolute intent as well. In League of Legends, its exact context filter provided optimal results.

Export Table Looping Predicate: To capture looping over the ET, a subsequent `0x18` offset dereference is added to the prior query. Its value, yielding the number of function names, must be used in a loop predicate to indicate a search over the size of the table. With lightweight context-filtering of *exe-only*, both ShadowHammer and Monju show absolute maliciousness, while *entry-only* for League of Legends is highly malicious.

Export Table Byte-Comparison: A clause was added to the ET query to indicate an attempt to find one of its exported functions through a string comparison. It is detected by first following the ET pointer to a loop-body scope that loads a character and performs a comparison against it. The results indicate absolute malicious intent under all three attacks using a single, generalized context of *no-exports* because benign binaries package this functionality in DLL-export functions.

Single/Multiple XOR-in-a-loop (XIAL): It is not uncommon for benign binaries to apply *XIAL* (e.g., encryption), and thus the challenge is to determine the suiting contexts evidencing malicious intent. For NetSarang, League of Legends, and CCleaner, seeking out a single *XIAL* instance within a function coupled with satisfactory context provides high maliciousness. Binaries across all file classes less frequently contain multiple instances of *XIAL* within a function. Monju makes use of this, and its context filter indicates highly malicious intent as well.

XOR-in-a-loop Sinks: This behaviour is an extension of *XOR-in-a-loop* that taint-follows the decrypted buffer to Win32 API sink function(s). The only attack performing this is Monju with sinks `LoadLibraryA`, `CreateFileA`, `CopyFileA`, and `WriteFileA`, considered in its query *XOR-in-a-loop sinks*$_{q1}$. By ignoring export functions and considering a depth up to 1, absolute maliciousness is achieved. At a depth of 2, a decrypted buffer sinks into `RegCreateKeyExA`. This variant yields absolute malicious intent, but a low sample count. We thus seek *any* sink (*XOR-in-a-loop sinks*$_{any}$), since this would still capture the Monju sample but with more detections. By excluding exports and considering depths up to 2, the filtering reveals 96.09% maliciousness and 246 detected samples.

Registry String Reference: Myanmar adds an executable to Windows startup through the registry via the "reg add" command. A query was constructed to search for strings with "reg [OPTION]". These strings indicate absolute maliciousness within 11 malware samples when no context filter is applied.

Sequential Constants: When ShadowHammer attempts to load modules `kernel32`, `ndll`, `IPHLAPI`, and `wininet`, they are first represented as strings stored as a sequence of hexadecimal constants in the stack. Thus, we reconstruct the strings from these constants and analyze their contents. The query captures each hex-constant that is assigned directly to a memory location. In post-processing, each constant is converted into its ASCII equivalent (e.g., `ASCII(0x646C6C) = 'dll'`) and concatenated into a single string. The module-string `'kernel32'` found in such *sequential constants* is highly indicative of malicious intent.

Module Filename Sink: Both Myanmar and Monju fetch their module path through `GetModuleFileName`, and sinks it into several functions; the behaviour indicating self-replication to a specific location for persistence. Myanmar sinks its path into $\texttt{CopyFileW}_{param1}$ and is captured with query q_1, while Monju sinks its path into both $\texttt{CopyFileA}_{param1}$ and $\texttt{CreateFileA}_{param1}$ considered in query q_2. In both cases, these sinks represent high indication of malware.

Reflective Loading: Development may require conditionally loading a module given certain system version/capability constraint(s). `LoadLibrary` can dynamically load a module for such a purpose, and its counterpart `GetProcAddress` reflectively retrieves a function pointer by its name within such a module. Malware also employs this technique to acquire function pointers with the goal of avoiding explicit, static function-call analysis. MonPass requested `'kernel32'` of `LoadLibrary`, from which several functions were reflectively loaded through `GetProcAddress`: `VirtualProtect`, `VirtualAlloc`, `CreateThread`, and `WaitForSingleObject`. A binary loading this module and *any* of the four functions in any context is highly indicative of malicious intent.

URL String: Monpass executes remote code fetched from a hard-coded HTTP URL over port 8880 containing a `.bmp` resource path. Similar URLs are sought from within the dataset, and no occurrences are found that connect to port 8880, or fetch a `.bmp`; thus, this form of URL is anomalous. Images (`bmp`, `jpeg`, or `png`) fetched over any port, however, are malicious (19 malware vs. 1 benign). As a catch-all behaviour, *any http string* found within the context of MonPass reveals a malicious intent of 86.57%, with only 29 benign false-positive samples.

Conditional Process Termination (CPT): Malware sniffs out anti-malware environments through characteristics like low RAM and disk space. This behaviour is encapsulated in a query that captures any Windows API with a buffer-parameter or return value reaching a predicate whose body contains a process-terminating call. MonPass exhibits this in two cases: one referring to RAM through a call to `GlobalmemoryStatusEx`, and the other to disk space through `DeviceIoControl` — both anomalous. As a catch-all behaviour, *any CPT* exhibited in MonPass' context proves highly malicious. Myanmar exhibits anomalous behaviour with `OpenEventA` leading to process termination as well. Applying *any CPT* over its context yields 85.6% maliciousness with 535 detected samples.

Parametric Momentum (PM): This metric, defined in Section 3, is applied to the entire dataset, computing a value for every function call and each of its parameters. These values were filtered post-analysis to assess the functions, parameters, and PM values seen in League of Legends. For example, `LoadLibrary` with PM = 21.0 and

$$
\begin{aligned}
&(\$,\$,\$) && \rightarrow 68.89\%,\ 1391\\
&(\$,!\,export,\$) && \rightarrow 73.17\%,\ 1293\\
&(\$,!\,export,[0,1]) && \rightarrow 95.29\%,\ 527\\
&(\$,!\,export,[1,1]) && \rightarrow 100.0\%,\ 348
\end{aligned}
$$

Fig. 6. NetSarang incremental context → P(Malware |Q), No. Samples.

above in its first parameter is denoted by $\texttt{LoadLibrary}_{p1\geq 21.0}$. This, and three other PMs are computed for League of Legends: $\texttt{GetProcAddress}_{p2\geq 10.25}$, $\texttt{VirtualAlloc}_{p1\geq 24.125}$, and $\texttt{VirtualAlloc}_{p1\geq 24.125\wedge p2\geq 21.125}$. These indicate maliciousness; `LoadLibrary` being the most effective with 95.69% at 222 samples. The other three variants score between 81 and 93%.

Executable Memory: Virtual memory is allocated by `VirtualAlloc`. It can be immediately assigned execution rights, or updated later via `VirtualProtect`. These functions may be called explicitly, or reflectively through `GetProcAddress`. With execution rights, virtual memory can be filled with a payload and executed, avoiding static detection.

NetSarang and League of Legends exhibit this behaviour with an explicit call to `VirtualAlloc`, denoted $\texttt{VirtualAlloc}_{explicit}$. The former attack, at a depth of 1 and ignoring exports, indicates absolute maliciousness with 348 samples; the latter follows as highly malicious with 1133 samples.

Myanmar and MonPass use `VirtualProtect` to set up execution rights; explicitly called in the former ($\texttt{VirtualProtect}_{explicit}$), and reflectively in the latter ($\texttt{VirtualProtect}_{reflective}$). Both reveal malicious intent with 97.97%, 100%, respectively. In the case of Myanmar, a lightweight depth of 1 is applied, while in MonPass, none.

Our initial hypothesis is that semantics contained in SSCAs carry evidence of malicious intent. By identifying, matching, and vetting the behaviours, we validate the hypothesis considering each attack contains highly malicious behaviour(s), and with the exception of CCleaner, all contained behaviours with absolute maliciousness. This indicates the samples could have been detected by their behaviour, however only if they were available in the past, an inquiry addressed in Section 4.3.

4.2. Behavior contextualization

The choice of data-flow behaviours comes with its pitfalls; though it may be malicious in some binaries, it can be used with benign intent in others. To account for this, behaviour context information is preserved and applied during statistical inference to accentuate malicious intent. This application of context is investigated in response to **RQ3 (Contextualization)**.

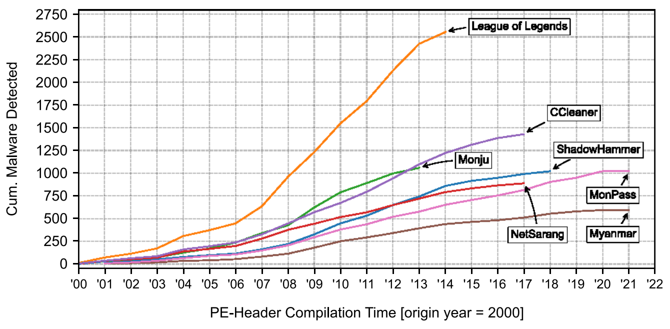


Fig. 7. Prevalence of SSCA attack behaviours.

NetSarang's *executable memory* is an example of the incremental benefits of context filtering, seen in Fig. 6. We notice an improvement of roughly 30% maliciousness over the applied contexts, with a tradeoff loss of detected samples. We observe similar improvements relative to the baseline context ($, $, $): 7.33% in Shadowhammer's *Export Table Offsets* behaviour, and 28.01% and 20.85% in NetSarang's and CCleaner's *Single XOR-in-a-loop* behaviour, respectively. The counterpart to acute context is its relaxation. NetSarang, for example, exhibits a behaviour within the initterm entry-point with absolute maliciousness but only 6 samples. A more relaxed context, however, yields 875 samples with ≈7% loss to intent.

As stated earlier, the nature of some behaviours makes them indistinguishably present in both benign and malware, and thus fruitless for detection. Yet, when combined with their context, their intent is brought to the surface. The majority of behaviours studied in Table 1 experienced such a boost in maliciousness.

4.3. Attack prevalence

The evidence presented in prior subsections confirm the presence of malicious behaviour(s) in every sample. To quantify the time available for defending against them, we answer **RQ4 (prevalence)**. To this end, we count the unique malware instances that exhibit SSCA behaviours, and whose compilation time comes before it. Fig. 7 represents the attack timeline, where each label represents the year of the attack, and its height above the x-axis, the cumulative number of past malicious samples containing its behaviour(s).

The SSCA samples have anywhere between 600 and 2500 binaries exhibiting their specific behaviour(s) prior to their attack date, with League of Legends leading as the most likely detectable sample, and Myanmar as the least likely to detect. It is important to observe that the behaviours are found in 3%–12.21% of the dataset, which is a representative of the set of binaries in the wild. Their presence dates back 13–21 years prior to each attack, and demonstrates that the behaviours of modern attacks are not necessarily novel, and in-fact recur over time. The prevalence plot is a long-term illustration that present efforts for capturing behaviours can support defense for years.

4.4. Query effectiveness

To prescribe the investigative approach in Section 3, results from the *statistical inference* should indicate a behaviour's malicious intent so that its future instances may be mitigated. However, as it tends to recur through time, the more detections made by its query, the more likely it will contribute to detecting future attacks. Equally important is a query's ability to prevent false-positive benign detections. Thus, responding to **RQ5 (effectiveness)**, the coverage and precision of queries determines their effectiveness. The benefit-cost ratio (BC) of each query from Table 1 is calculated via $BC_q = \frac{True\ Positives}{max(1,\ False\ Positives)}$ to evaluate effectiveness.

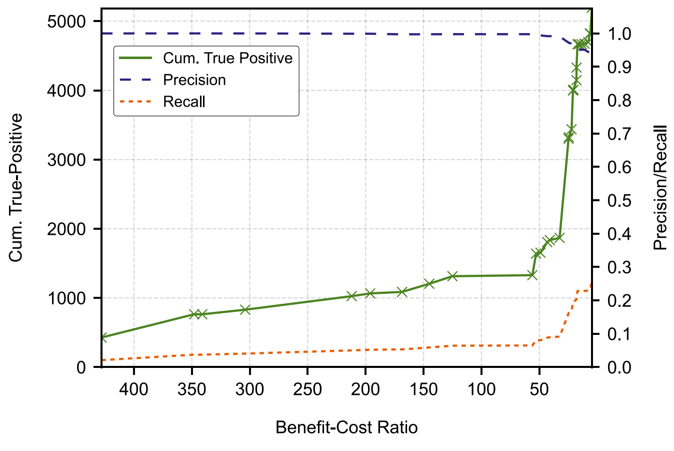


Fig. 8. Benefit-Cost of queries applied over dataset.

Each computed ratio is plotted against the unique cumulative true-positive detections in Fig. 8, where each *x* marker represents a query. The highest benefit queries contribute to a precision of 100%, while the lower benefit queries are responsible for a drop to 92.8%. From this, we note queries can generate precise results, addressing the concern of excessive benign false alarms. To illustrate strong detection coverage, we note that only 15 behaviours from seven samples are responsible for matching a quarter (recall of 25.25%) of the malware dataset.

Further evidence of query effectiveness comes from our novel metric, parametric momentum. With benign instances of $\texttt{WinExec}_{p1}$ peaking at PM = 12.0, if the benign-maximum PM becomes a high-pass filter threshold (e.g., PM > 12.0), the 134 malicious instances that exceed this value are detections free of false positives. Applying this technique to the entire dataset produces substantial coverage (2600 samples, 12.71%).

Fifteen behaviours derived from seven SSCA samples yield high precision, and cover a quarter of the malware dataset. In addition, parametric momentum covers over one-tenth without false positives. Though this work aims at providing an investigative study of SSCAs using their query-encodable behaviours, the above demonstrates the effectiveness of queries as a promising, yet-to-be-investigated mechanism within a detection system thanks to their considerable coverage, and low false-positive alarms.

5. Related work

To our knowledge, little work has been done regarding SSCAs at the binary level, thus we summarize open-source works. Prior works propose solutions to address SSCAs, but none are full-proof. Torres-Arias et al. (2019) propose *in-toto* which builds cryptographic integrity into the software process, and assumes private keys are inaccessible to attackers. *Guix*, assembled by Courtès (2022) is a tool that removes dependence on opaque binaries by rebuilding them from authenticated source-code, but encounters resistance in merging unauthenticated contributions. The SLSA Committee (2022) put forward *SLSA*, a framework of standards for protecting against SSCAs through levels of integrity guarantees. Lamb and Zacchiroli (2022) develop reproducible builds that promise bit-for-bit integrity of binaries built from source via a consensus over trusted checksums, but does not guarantee the source is untampered.

Anomalicious detects malicious commits through anomalous repository changes like LOC removed, but ignores code semantics (Gonzalez et al., 2021). Garrett et al. (2019) scan package repositories for malicious system calls and package-dependencies across updates through anomaly detection models of varied patch-granularity. Duan et al. (2021) present *MalOSS*, a tool that claims to identify malicious packages by comparing behavioural history of software updates to spot outlier data-flows. Another tool, *PPD*, flags malicious packages in PyPI based on anomaly detection of AST and regular expression-based features (Liang et al., 2021). SIGL is a proposed method of detecting malware during software installation (Han et al., 2021). Features extracted from an installation are run through an autoencoder to spot anomalies, however, it would miss attacks activated post-installation (e.g., ShadowHammer).

Several studies propose best practices and paradigm shifts for remediating the software supply chain attack problem. Collier and Sarkis (2021) advocate for a zero trust policy with the onus of safety resting on each client, but such a policy is arguably infeasible for enterprises without the necessary resources. NIST (2015) takes a similar view in their “best practices”, suggesting a strict assessment of supplier security posture. Ohm et al. (2020) suggest that software updates be considered a potential attack vector, and recommend version pinning; making security audited versions permanent.

An empirical study by Zahan et al. (2022) on the `npm` supply-chain support use of package metadata for flagging suspicious software through weak-link signals such as presence of install scripts, and unmaintained packages. A large collective review by Ohm et al. (2020) find Node.js and Python packages execute malicious code either within installation scripts or at runtime, and may be conditionally executed; the work does not describe an approach for thwarting such attacks.

6. Conclusion

Binary code does not provide the luxury of verifying the trustworthiness of its sources. Rather than heedlessly trusting such code, we propose an approach that is effective at unearthing its malicious intent through static flow analysis.

6.1. Findings

The SSCA samples carried behaviours with statistically-demonstrable malicious intent (86.57–100%), present in the wild between 13 and 21 years prior to each attack based on a representative dataset. This may indicate that these attacks could have been thwarted through static analysis and proper contextualization. We find this last mechanism can boost the intent of a matched behaviour by 30%. Novel metrics such as parametric momentum have promising detection capabilities, detecting 12.71% of our dataset without false positives. Overall, the few behaviours observed from real SSCAs matched 25.25% of malware with 92.8% precision, a manifestation of detection effectiveness, and a mark of its utility within a detection system.

This paper presents a perspective of complete distrust in the release-binary, and relies on identifiable behaviours to assess its safety. This is in contrast to the works of Section 5 that cover attacks on OSS. Such solutions rely on source-integrity and the OSS community for code cleanliness, yet neglect code semantics that we have shown may carry malicious intent. To this regard, querying the release binary can act as an additional line of defense.

6.2. Future work

In its current state, queries match data-flow expressions intra-procedurally. An avenue for extension would be scoping these interprocedurally. Considering indirect calls could expand the call-graph for better coverage, while path feasibility could validate paths. Deeper investigation into novel context information is of interest, for example: call-graph dominators and cycle-members, or conditional-branches. Numerical and categorical features derived from queries, such as *parametric momentum*, can be used as input to machine learning classifiers.

Automatic translation of a behaviour into a query would be an asset to the reverse engineering process that would incentivize concretizing queries for future attack-prevention. The abstraction of low-level analysis through high-level expressions would provide freedom to practitioners to question binary code semantics. We look to forward this mechanism with an intuitive interface, comprehensive capabilities, and customizable analysis features for advancing the state of the art of binary analysis.

References

Beltov, M., 2017. Netsarang apps riddled with shadowpad backdoor. https://sensorstechforum.com/netsarang-apps-shadowpad-backdoor/.

Brand, M., Valli, C., Woodward, A., 2010. Malware forensics: discovery of the intent of deception. J. Digit. Forensics Secur. Law 5, 31–42.

Brumaghin, E., Carter, E., Mercer, W., Molyett, M., Olney, M., Rascagneres, P., Williams, C., 2017a. Ccleaner Command and Control Causes Concern. https://blog.talosintelligence.com/2017/09/ccleaner-c2-concern.html.

Brumaghin, E., Gibb, R., Mercer, W., Molyett, M., Williams, C., 2017b. Ccleanup: a vast number of machines at risk. https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html.

Camastra, L., Morgenstern, I., Vojtěšek, J., 2021. Backdoored client from Mongolian ca monpass. https://decoded.avast.io/luigicamastra/backdoored-client-from-mongolian-ca-monpass/.

Chappell, G., 2016. Peb_ldr_data. https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntpsapi_x/peb_ldr_data.htm.

Cimpanu, C., 2021. Backdoor malware found on the Myanmar president's website, again. https://therecord.media/backdoor-malware-found-on-the-myanmar-presidents-website-again/.

Collier, Z.A., Sarkis, J., 2021. The zero trust supply chain: managing supply chain risk in the absence of trust. Int. J. Prod. Res. 59, 3430–3445.

Context Threat Intelligence, 2014. Context threat intelligence, threat advisory: the monju incident. https://samples.vx-underground.org/APTs/2014/NoMonth/TheMonjuIncident.pdf.

Courtès, L., 2022. Building a Secure Software Supply Chain with GNU Guix. The Art, Science, and Engineering of Programming, vol. 7.

CyberSecurityHelp, 2021. Supply Chain Attack Targets myanmar Presidential Office Website. https://www.cybersecurity-help.cz/blog/2146.html.

Damaye, S., 2017. Pe portable executable: export table. https://www.aldeid.com/wiki/PE-Portable-executable#Analyze_PE_files.

Duan, R., Alrawi, O., Kasturi, R., Elder, R., Saltaformaggio, B., Lee, W., 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages.

Executive Office of the President, 2021. Improving the nation's cybersecurity. Executive Order 14028. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity.

Falcone, R., 2015. Evilgrab delivered by watering hole attack on president of Myanmar's website. https://unit42.paloaltonetworks.com/evilgrab-delivered-by-watering-hole-attack-on-president-of-myanmars-website/.

Ferguson, S., 2019. shadowhammer” spreads across online gaming supply chain. https://www.bankinfosecurity.com/shadowhammer-spreads-across-online-gaming-supply-chain-a-12409.

FIREEYE, 2020. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor. https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.

GameRant, 2015. Malware detected in 'league of legends' & 'path of exile'. https://gamerant.com/league-legends-path-exile-hacked-files/.

Garrett, K., Ferreira, G., Jia, L., Sunshine, J., Kästner, C., 2019. Detecting suspicious package updates. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results. ICSE-NIER), pp. 13–16.

Gonzalez, D., Zimmermann, T., Godefroid, P., Schäfer, M., 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. IEEE Press.

GREAT, 2017. Shadowpad in corporate networks. https://securelist.com/shadowpad-in-corporate-networks/81432/.

GREAT, A.M.R., 2019. Operation shadowhammer. https://securelist.com/operation-shadowhammer/89992/.

Han, X., Yu, X., Pasquier, T., Li, D., Rhee, J., Mickens, J., Seltzer, M., Chen, H., 2021. SIGL: securing software installations through deep graph learning. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 2345–2362.

Herr, T., Lee, J., Loomis, W., Scott, S., 2020. Breaking trust: shades of crisis across an insecure software supply chain. Technical Report. https://www.atlanticcouncil.org/wp-content/uploads/2020/07/Breaking-trust-Shades-of-crisis-across-an-insecure-software-supply-chain.pdf.

Hyvärinen, N., 2019. Analysis of shadowhammer asus attack first stage payload. https://blog.f-secure.com/analysis-shadowhammer-asus-attack-first-stage-payload/.

Kaspersky, 2017. Shadowpad: popular server management software hit in supply chain attack. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2017/08/07172148/ShadowPad_technical_description_PDF.pdf.

Lamb, C., Zacchiroli, S., 2022. Reproducible builds: increasing the integrity of software supply chains. IEEE Software 39, 62–70.

Liang, G., Zhou, X., Wang, Q., Du, Y., Huang, C., 2021. Malicious packages lurking in user-friendly python package index. In: 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications. TrustCom), pp. 606–613.

López, G., 2020. Component object model hijacking. https://attackiq.com/2020/03/26/component-object-model-hijacking/.

Malvica, M.. [windows] exploit dev. https://www.matteomalvica.com/minutes/exploit_dev/.
Microsoft, 2022. Malware names. https://docs.microsoft.com/en-us/microsoft-365/security/intelligence/malware-naming?view=o365-worldwide.
MITRE, 2021. Obfuscated files or information: binary padding. https://attack.mitre.org/techniques/T1027/001/.
NIST, 2015. Best practices in cyber supply chain risk management. https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/briefings/Workshop-Brief-on-Cyber-Supply-Chain-Best-Practices.pdf.
NIST, 2022. Improving the nation's cybersecurity: NIST's responsibilities under the may 2021 executive order. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity.
Ohm, M., Plate, H., Sykosch, A., Meier, M., 2020. Backstabber's knife collection: a review of open source software supply chain attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer.
Ryansecurity, 2015. Plugx malware found in official releases of league of legends path of exile. https://herrymorison.tistory.com/entry/PlugX-Malware-Found-in-Official-Releases-of-League-of-Legends-Path-of-Exile.
SLSA Committee, 2022. Supply chain levels for software artifacts (SLSA). https://slsa.dev/spec/v0.1/.
the Robot, M., 2019. Shadowhammer: new details. https://www.kaspersky.com/blog/details-shadow-hammer/26597/.
Torres-Arias, S., Afzali, H., Kuppusamy, T.K., Curtmola, R., Cappos, J., 2019. in-toto: providing farm-to-table guarantees for bits and bytes. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, pp. 1393–1410.
Trend Micro, 2015. Gamers beware: league of legends, path of exile released with data-stealing malware. https://www.trendmicro.com/vinfo/it/security/news/cybercrime-and-digital-threats/league-of-legends-path-of-exile-released-with-data-stealing-malware.
Vlček, O., 2018. Ccleaner apt attack: a technical look inside, RSAConference. https://www.rsaconference.com/library/presentation/ccleaner-apt-attack-a-technical-look-inside.
WeiRanLab, 2018. Xshell backdoor analysis. https://isecurity.huawei.com/sec/web/viewBlog.do?id=1895.
Zahan, N., Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., Williams, L., 2022. What are weak links in the npm supply chain?. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE Computer Society, Los Alamitos, CA, USA, pp. 331–340.